Studio Modeling Platform™

MQL Guide

3DEXPERIENCE R2017x

Table of Contents

Preface

The *Studio Modeling Platform MQL Guide* is intended for users who:

- use a command driven interface to all of the functions that can be performed in 3DSpace.
- use MQL to perform most of the operations that can be done using Matrix Navigator or Business Modeler.
- use MQL to perform other functions of 3DSpace that you can not do with the GUI applications, including system administration functions.

About This Guide

Purpose

This guide provides the MQL programmers with conceptual and reference information concerning the MQL language.

MQL is the Matrix Query Language. Similar to SQL, MQL consists of a set of commands that help the administrator set up and test a 3DSpace database quickly and efficiently.

MQL is primarily a tool for building the 3DSpace database. Also, you can use MQL to add information to the existing database, and extract information from the database.

This guide provides MQL programming concepts to persons responsible for building and maintaining the 3DSpace database.

Intended Audience

You can use MQL to perform most of the operations that can be done using Matrix Navigator or Business Modeler. MQL also provides additional commands for functions that can't be done with these GUI applications. MQL commands can be run on a command line or via a script. For these purposes, you can review the introduction and concepts chapters of this guide, or the following related documentation:

- *Matrix Navigator Guide*

  Review this guide to become familiar with the Matrix application.
- *Business Modeler Guide*

  Review this guide to become familiar with the Business Modeler application, and the kinds of tasks the Business Administrator performs.

Skills You Need

Programmers who will use MQL in Java or.Net programs to be used across the enterprise should have a strong foundation in the following skills:

- Knowledge of database management and Web servers
- Object-oriented programming techniques
- Programming in a multi-threaded environment
- Java, JavaServer Pages, and JavaScript or C# and ActiveServer Pages programming experience.

Introducing 3DSpace

An Information Management System

3DSpace is a standards-based, open and scalable system able to support the largest, most complex, product lifecycle management deployments. It provides the flexibility to easily configure business processes, user interfaces, and infrastructure options to ensure that they meet your organization's needs. The system enables you to continually drive business process improvements to operate more efficiently using pre-built metrics reports, while virtual workplace capabilities enable ad-hoc collaboration for cross-functional and geographically dispersed teams to securely share product content.

The 3DSpace system operates in virtually any configuration to support your unique operating, organizational, and performance needs—on a single computer, in a networked system environment, over the internet or an enterprise intranet.

This chapter introduces you to the concepts and features of 3DSpace.

3DSpace Database Components

3DSpace (CPF) provides the backbone for product lifecycle management activities within a workgroup, enterprise, or extended enterprise, depending on the organization's needs. 3DSpace is the underlying support for all 3DEXPERIENCE products and other Dassault Systèmes products for Product Lifecycle Management, which include products from CATIA, 3DLive, SIMULIA, and DELMIA product lines. It is comprised of the following end-user software components:

- *Matrix Navigator* is the tool for searching the data objects based on the data model. It is installed as part of the Studio Modeling Platform Rich Clients distribution. The CPF license entitles your use of this component of Studio Modeling Platform only.
- *3DSpace Service* is a Java/RMI-based business object server. The MQL command line executable is included with the server for the purpose of system installation and data/file storage setup functions only. If any other schema configurations are needed, Studio Modeling Platform (DTE) must be licensed. Note that if distributed file stores are created with Sites and Locations, one or more File Collaboration Server licenses may also be needed.
- *Collaboration and Approvals* include the capabilities from the Application Exchange Framework, Common Components, Team Central, and Business Process Metrics applications.

The following administrative add-ons are also available:

- *Studio Modeling Platform (DTE)* provides the development tools for a company to define and test configurations that are needed in their production system. While the 3DEXPERIENCE platform offers PLM products that cover many product development business processes, the need for companies to tailor or extend products to meet their specific needs is inevitable, since a company's competitive advantage often requires a unique product development process compared to how other companies execute. Therefore, in order to deploy the 3DEXPERIENCE system, companies may need to set up an environment that allows them to develop changes to the standard products and test them in conditions that duplicate the actual or projected network and server performance. The Studio Modeling Platform is required in order to tailor and configure the production system.

  Users of Studio Modeling Platform must be valid licensees of the products that are configured and tested. However, a user's production license can also be used with the system installed for Studio Modeling Platform. The Studio Modeling Platform includes the following primary tools:

    - MQL - the command line interface tool for executing commands and scripts.
    - System Administration - the tool for managing and configuring data and file storage vaults.
    - Business Modeler - the tool for managing the data model and its associated business processes including types, attributes, relationships, and the web user interface design.
    - Matrix Navigator - the tool for searching the data objects based on the data model. This application is required to administer triggers and other configuration objects.While the Matrix Navigator is installed as part of Studio Modeling Platform for ease of installation, this application license is included in both CPF and DTE. For legacy customers that are still based on thick client deployments vs. Web, the CPF license entitles you to use Matrix Navigator.
- *Studio Federation Toolkit* (ADT), which provides documentation and examples for writing custom programs that use the Adaplet libraries available in 3DSpace. The Studio Federation Toolkit must be licensed for each Studio Modeling Platform environment that requires development of custom connectors with the Adaplet APIs.
- *File Collaboration Server* (FCS), which enables administrators to distribute file data across the enterprise for optimal upload and download performance. A system license is required for each physical site that requires remote file storage. There is no limit to the number of users that can use the File Collaboration Server at a given licensed site, however, all of these users must be licensed to use 3DSpace, which is the prerequisite.

Overview of Business Process Services

Business Process Services (BPS) is the foundation for all other apps. It includes the schema for all products, as well as the programs and JavaServer Pages needed to construct the user interface shared by all the applications. It can also be used as the basis for creating your own applications.

BPS (and the other components and prerequisites) must be installed before you can install any other apps. Refer to the B*usiness Process Services Installation Guide* for details.

Business Process Services Components

Application Components

Each Dassault Systemes product contains the items listed in this table.

Application Components	
Item	**For information, refer to:**
Web pages used by the application's users	The user guide that accompanies the application.
Programs specific to the application	To configure programs and for descriptions of utility trigger programs, see *Business Process Services Administration Guide: Triggers*. For application-specific trigger programs, see the Administration Guide that accompanies the application. For information on how to call the included JavaBeans in your custom applications, see the Javadocs located at: ENOVIA_INSTALL\Apps\APP_NAME\VERSION\Doc\javadoc. Refer to *Directories for shared and common components* for other details.
Other administrative objects specific to the application, such as formats	The Administration Guide for the application.
Business objects that accomplish system-related tasks, such as objects for automatically-naming objects and for executing trigger programs	For general information on how the objects function and how to configure them, see the *Business Process Services Administration Guide: Configuring Business Process Services Functions*. For a list of the objects included in the application, see the Administration Guide that accompanies the application.

Directories for shared and common components

Some applications, such as Materials Compliance Central, install other components that may be shared between applications. When this is the case there are 2 directories installed under ENOVIA_INSTALL\Apps, one which includes "base" in the name, such as MaterialsComplianceBase.

Use of General Client Applications

Some of the instructions in this and other Administration Guides require the use of a general Matrix client navigator. It is important to restrict the use of these general navigator applications to only a few specially-trained business administrators.

The general client navigator is the desktop version of Matrix Navigator (also known as the thick client), including MQL, Matrix, and Business Modeler.

It is important to restrict the use of these general navigator apps to only a few specially-trained business administrators and to only the purposes described in the *Business Process Services - AEF User Guide* and app Administration Guides. Apps run JavaBean code that requires data to have specific characteristics and conditions. For example, objects may have to have certain relationships defined, have specific values entered for attributes, be in specific lifecycle states, or be in particular vaults. When a person works within the app user interface, these data conditions are met. However, the general Matrix navigators are not necessarily aware of these conditions and therefore a person working within the general navigators can easily compromise data integrity.

Another reason to restrict access to the general clients is that certain actions have different results depending on where the action is taken. A command on a JSP page may include options (such as additional MQL clauses) to ensure that the operation is completed as the application expects, but a user in a general client has no guidance on what options should be chosen. For example, when a file is checked into 3DSpace using a general client, the store set in the policy is used; when using an app to check in a file, the person or company default store is used regardless of the store set by the policy.

The general navigators must or can be used in situations such as:

- App features require data that cannot be created within the user interface.

  For example, some user profile information and template information must be created in a general navigator.
- Automated business rules and processes need to be configured, such as triggers and autonamers.
- Data needs to be investigated for troubleshooting, testing, or data conversion.

The general navigators should only be used in these situations, using the instructions provided in documentation, and only by specially-trained business administrators. Standard users of apps should never be allowed to work with their data in a general navigator and external customers should never be given access to a general navigator.

Related Documentation

This section lists the documentation available for BPS and 3DEXPERIENCE apps.

- *Studio Modeling Platform Overview*
- *MQL Command Reference*
- *BPS Administrator Documentation*
- *App Product Administrator Documentation*
- *Related Documentation Not Installed with BPS or Applications*

Studio Modeling Platform Overview

In the online documentation, this guide introduces the concepts and features of the 3DEXPERIENCE platform. This guide includes information about building the system basics (vaults, stores, locations, and sites).

MQL Command Reference

In the online documentation, this reference provides detailed syntax and descriptions for all the MQL commands.

BPS Administrator Documentation

BPS installs with this documentation:

- *Business Process Services Administration Guide*

  This guide is available in HTML format. It is for people in the host company who need to configure and customize apps. It describes the schema that underlies the applications and how to configure it.
- JavaDoc for BPS

  For descriptions of methods in framework packages and classes, see `ENOVIA_INSTALL\Apps\Framework\VERSION\Doc\javadoc.`

App Product Administrator Documentation

Each app has a separate guide for company administrators who work with 3DEXPERIENCE apps. These are the same people who will use BPS and the Studio Modeling Platform to configure an app. The Administration Guide for each app contains information that is unique to the app and therefore not appropriate for the *Business Process Services Administration Guide*. The Administration Guides are provided in PDF format.

Related Documentation Not Installed with BPS or Applications

A Program Directory is available for each version of BPS and the 3DEXPERIENCE platform apps. Each version comes with media that includes the program directory for that release. The program directory is a website that organizes all the release information for all Dassault Systèmes products for a given release. It contains information about prerequisites, installation, licensing, product enhancements, general issues, open issues, documentation addenda, and closed issues.

For instructions on installing BPS and applications, see the *Business Process Services Installation Guide*.

How to Use this Guide

This guide explains the concepts and reference material you need to setup the 3DSpace database and also to write applications for it. This guide is organized in conceptual chapters and provides complete syntax and reference information in the latter sections:

- Chapter 1, *The MQL Language* - Introduces the MQL Language, syntax conventions, scripts, input modes, commenting, explains how to build an initial database and edit an existing one.
- Chapter 2, *Working with Metadata* - Provides information about how to define meta data. It describes attributes, dimensions, types, interfaces, formats, relationships, and policies.
- Chapter 3, *Manipulating Data* - Explains how to use business objects to manipulate data.
- Chapter 4, *Working with Workspace Objects* - Explains how to use workspace objects to manipulate data. It provides information on queries, sets, filters, and cues.
- Chapter 5, *Extending an Application* - Explains how to use other features in the system to extend an application such as programs, and applications.
- Chapter 6, *Modeling Web Elements*- Provides information about commands, menus, tables, portals, etc.
- Chapter 7, *Maintenance* - Provides information about how to maintain the system. There are sections on monitoring clients, system-wide settings, database constraints, diagnostics, etc.
- Chapter 8, *Working with Import and Export* - Discusses concepts about exporting and importing administrative and business objects.

Published examples in this document, including but not limited to scripts, programs, and related items, are intended to provide some assistance to customers by example. They are for demonstration purposes only. It does not imply an obligation for Dassault Systemes to provide examples for every published platform, or for every potential permutation of platforms/products/versions/etc.

What To Do Next

Start by understanding the basic concepts explained in the chapters of this guide and then use the reference as a tool when programming in MQL. The reference section will provide complete syntax and description of all the parameters, clauses, and grammar of MQL commands.

You can also use MQL within Java or .Net programs when using the MQL Command package of the Studio Customization Toolkit. Before you begin programming for 3DSpace, you should review the following information:

- *3DSpace Installation Guide*

  The 3DSpace Service must be installed before installing Studio Customization Toolkit. Information on installing the server is included in the this guide.

Always refer to the current Program Directory for any changes that have been made since the publication of this manual.

1 The MQL Language

This chapter introduces the MQL command language in the following sections:

- *About MQL*
- *Accessing MQL*
- *Using Commands and Scripts*
- *Building an Initial Database*
- *Modifying an Existing Database*
- *Working with Transactions*
- *Working With Threads*
- *Using Tcl*

About MQL

The Matrix Query Language (MQL) is similar to SQL. MQL consists of a set of commands that help the administrator set up, build, and test a 3DSpace database quickly and efficiently.

You can also use MQL to add information to the existing database, and extract information.

MQL acts an interpreter for 3DSpace and can be used in one of three modes:

1. Interactive Mode - which means executing each command from the command line.This mode is typically used when you have only a few modifications to make or tests to perform.
2. Script Mode - which means using scripts to run commands. This lets you combine commands and also have a repeatable history of commands. You should use MQL scripts as long as you are in the building process. Later, you may decide to add information and files into 3DSpace. When adding information you may require additional MQL commands. Rather than entering them interactively, you can create a new script to handle the new modifications.
3. Tool Command Language (Tcl) Mode - which means you can use the tcl/tk scripting language. With Tcl embedded in MQL, common programming features such as variables, flow control, condition testing, and procedures are available.

After the database is built, you will most likely maintain it through the Business Modeler interface which helps you see what you are changing.

Using Interactive Mode

When you use MQL in interactive mode, you type in one MQL command at a time. As each command is entered, it is processed. This is similar to entering system commands at your terminal.

MQL is not intended to be an end-user presentation/viewing tool, and as a consequence there are cases where some non-ASCII character sets (eg.some Japanese characters) will have characters that do not display properly, such as when a history record is printed to the console in interactive mode. This is due to low-level handling of the byte code when attempting to display. However, the data is intact when retrieved in any programmatic way, such as:

- Retrieving data into a tcl variable: set var sOut [mql print bus T N R select history]
- Retrieving data from a Java program via ADK calls.
- Writing data into a file: print bus T N R select history output d:/temp/TNR_History.txt;

Interactive mode is useful if you have only a few commands to enter or want more freedom and flexibility when entering commands. However, interactive mode is very inefficient if you are building large databases or want to input large amounts of information. For this reason, you also have the ability to use MQL in a script mode.

Command Line Editing

Command line editing is available in interactive MQL. MQL maintains a history list of commands and allows you to edit these commands using the arrow keys and some control characters.

To edit, use the arrow keys and move your cursor to the point where a change is required. Insert and delete characters and/or words, as needed.

Using Control Characters

All editing commands are control characters which are entered by holding the Ctrl or Esc key while typing another key, as listed in the following table. All editing commands operate from any place on the line, not just at the beginning of the line.

Command Line Editing Control Characters

Control Character	Description
Ctrl-A	Moves the cursor to the beginning of the line.
Ctrl-B	Moves the cursor to the left (back) one column.
Esc-B	Moves the cursor back one word.
Ctrl-D	Deletes the character to the right of the cursor.
Ctrl-E	Moves the cursor to the end of the line.
Ctrl-F	Moves the cursor right (forward) one column.
Esc-F	Moves the cursor forward one word.
Ctrl-H	Deletes the character to the left of the cursor.
Ctrl-I	Jumps to the next tab stop.
Ctrl-J	Returns the current line.
Ctrl-K	Kills from the cursor to the end of the line (see Ctrl-Y).
Ctrl-L	Redisplays the current line.
Ctrl-M	Returns the current line.
Ctrl-N	Fetches the next line from the history list.
Ctrl-O	Toggles the overwrite/insert mode, initially in insert mode.
Ctrl-P	Fetches the previous line from the history list.
Ctrl-R	Begins a reverse incremental search through the history list. Each printing character typed adds to the search substring (which is empty initially). MQL finds and displays the first matching location. Typing Ctrl-R again marks the current starting location and begins a new search for the current substring.
Type Ctrl-H	Or press the Del key to delete the last character from the search string. MQL restarts the search from the last starting location. Repeated Ctrl-H or Del characters, therefore, unwind the search to the match nearest to the point where you last typed Ctrl-R or Ctrl-S (described below).Type Esc or any other editing character to accept the current match and terminate the search.
Type Ctrl-H	Or press the Del key until the search string is empty to reset the start of the search to the beginning of the history list. Type Esc or any other editing character to accept the current match and terminate the search.
Ctrl-S	Begins a forward incremental search through the history list. The behavior is like Ctrl-R but in the opposite direction through the history list.
Ctrl-T	Transposes the current and previous character.
Ctrl-U	Kills the entire line (see Ctrl-Y).

Control Character	Description
Ctrl-Y	Yanks the previously killed text back at the current location.
Backspace	Deletes the character left of the cursor.
Del	Deletes the character right of the cursor.
Return	Returns the current line.
Tab	Jumps to the next tab stop.

The MQL Prompts

Interactive MQL has two prompts. The primary prompt is, by default:

```
MQL<%d>
```

where %d is replaced with the command number. The secondary prompt is, by default:

```
>
```

The secondary prompt is used when a new line has been entered without terminating the command.You can change the primary and secondary prompts with the MQL command:

```
prompt [[PROMPT_1] [PROMPT_2]]
```

Without arguments, the prompts reset to the defaults.

Using Script Mode

When working in the script (or batch) mode, you use a text editor to build a set of MQL commands contained in an external file (a script), that can be sent to the command interface. The interface then reads the script, line by line, and processes the commands just as it would in the interactive mode.

Working in script mode has many advantages, particularly when you are first building a database, such as:

- It gives you a written record of all your definitions and assignments. This enables you to review what you have done and make changes easily while you are testing the database. When you are first building the database, you may experiment with different definitions to see which one works best for your application. A written record saves time and aggravation since you do not have to print definitions when there is a question.
- You can use text editor features to duplicate and modify similar definitions and assignments. Rather than entering every command, you can copy and modify only the values that must change. This enables you to build large and complicated databases quickly.
- Testing and debugging the database is simplified. MQL databases can be wiped clean so that you can resubmit and reprocess scripts. This means you can maintain large sections of the MQL commands while easily changing other sections. Rather than entering commands to modify the database, you can simply edit the script. If you are dissatisfied with any portion of the database, you can change that portion of the script without eliminating the work you did on other portions. Since the script contains the entire database definition and its assignments, there is no question as to what was or was not entered.

Tcl Overview

Tcl (tool command language) is a universal scripting language that offers a component approach to application development. Its interpreter is a library of C procedures that have been incorporated into MQL.

With Tcl embedded in MQL, common programming features such as variables, flow control, condition testing, and procedures are available for use. The Tk toolkit is also included. Tcl and Tk are widely available and documented. The information in this section is simply an overview of functionality. For more detailed procedures, consult the references listed in the section.

Tcl enhances MQL by adding the following functionality:

- Simple text language with enhanced script capabilities such as condition clauses
- Library package for embedding other application programs
- Simple syntax (similar to sh, C, and Lisp):
  Here are some examples of using tcl

Command	Output
set a 47	47

Substitutions:

Command	Output
set b $a	47
set b [expr $a+10]	57

Quoting:

Command	Output
set b "a is $a"	a is 47
set b {[expr $a+10]}	[expr $a+10]

- Variables, associative arrays, and lists
- C-like expressions
- Conditions, looping

```
if "$x<<3" {
   puts "x is too small"
}
```

- Procedures
- Access to files, subprocesses
- Exception trapping

MQL/Tcl offers the following benefits to 3DSpace users:

- Rapid development—Compared with toolkits where you program in C, there is less to learn and less code to write. In addition, Tcl is an interpreted language with which you can generate and execute new scripts on the fly without recompiling or restarting the application. This enables you to test and fix problems rapidly.
- Platform independence—Tcl was designed to work in a heterogeneous environment. This means that a Tcl script developed under Windows can be executed on a UNIX platform. After you complete design and testing, a program object can be created using the code providing users with immediate access to the new application. This also solves the problem of a need to distribute new applications.
- Integration support—Because a Tcl application can include many different library packages, each of which provides a set of Tcl commands, an MQL/Tcl script can be used as a communication mechanism to allow different applications to work with 3DSpace.
- User convenience— MQL commands can be executed while in Tcl mode by preceding the correct MQL syntax with mql.

Tcl is described in detail at the end of this chapter.

Accessing MQL

There are several ways to access the MQL from the operating system.

For example, if you are working on a PC, select the icon.

Or

If you are working on a UNIX platform, enter the MQL command using the following syntax:

```
mql [options] [-] [file...]
```

Brackets [] *indicate optional information. Do not include the brackets when you enter the command.* **Braces** {} *may also appear in a command syntax indicating that the items in the braces may appear one or more times.*

When the system processes the mql command, the command interface starts up in one of two modes: interactive or script. The mode it uses is determined by the presence of the hyphen and file name(s):

Interactive mode is used when the command includes the hyphen and/or does not include a file name(s). For example, to run MQL interactively, enter either:

```
mql -
```

Or:

```
mql
```

In both of these commands, no files are specified so the interpreter starts up in an interactive mode to await input from the assigned input device (usually your terminal).

MQL Syntax	Specifies that...
mql	The MQL interpreter should be invoked.
options	One or more MQL command options
- (the hyphen)	Entries come from standard input which is interactive mode.
file	A script will be used. File is the name of the script(s) to be processed. 3DSpace processes the script of MQL commands and then returns control to the operating system. For example, `mql TestDefinitions.mql TestObjects.mql` This command invokes the MQL interpreter, processes the MQL commands within the TestDefinitions.mql script, and then processes the commands within the TestObjects.mql script.

MQL Command Options

In addition to the hyphen and file name qualifiers, several MQL command options are available:

mql Command Options	Specifies that MQL…
-b FILENAME	Use the bootfile FILENAME instead of matrix-r.
-c "command;command... "	Use command;command… as the input script. Processes the MQL commands enclosed within the double quotes.
-d	Suppress the MQL window but do not suppress title information.
-install -bootfile BOOTFILE -user DBUSER -password DBPASSOWORD -host CONNECTSTRING -driver DRIVER	Creates a bootfile with the parameters passed.
-k	Does not abort on error. Continues on to the next MQL command, if an error is detected in an MQL command. The -k option is ignored for interactive mode.
-q	Set Quote on.
-t	Suppress the printing of the opening title and the MQL window.
-v	Work in verbose mode to print all generated messages.

Each MQL command and command option is discussed in the reference section.

- FILENAME is the name of the file to be created.
- DBUSER is the name used to connect to the database. This is the database user that is created to hold the database. It is established by the database administrator.
- DBPASSWORD is the security password associated with the Username. This is established by the database administrator. The user password is encrypted as well as encoded.
- CONNECTSTRING For Oracle, the connect string is the Oracle “connect identifier”, which can be a net service name, database service name, alias or net service alias. For SQL Server, it is the name of the ODBC datasource.
- DRIVER is one of the following depending on which database you use:
  - Oracle/OCI80
  - CLI
  - MSSQL/ODBC

Using Commands and Scripts

All commands perform an action within 3DSpace. For example, actions might define new structures within the database, manipulate the existing structures, or insert or modify data associated with the database structures.

```
An MQL script is an external file that contains MQL commands (see the
example below). You can create this file using any text editor. After
you insert all the desired MQL commands, you can batch process them by
specifying the name of the script file in the mql command.
############################################################
# Script file: SampleScript.mql
# These MQL commands define a vault and create an object.
############################################################
verbose on;
#
# Set context to Matrix System Administrator's and define a new vault.
#
set context dba;
add vault "Electrical Parts";
output "Electrical Parts vault has been added"
#
# Set context to that of person Steve in vault "Electrical
Subassemblies"
#
set context vault "Electrical Subassemblies" user Steve;
#
# Add a business object of type Drawing with name C234 A3
# and revision 0 and check in the file drawing.cad
#
output "now adding a business object for Steve";
add businessobject Drawing "C234 A3" 0
   policy Drawing
   description "Drawing of motor for Vehicle V7";
checkin businessobject Drawing "C234 A3" 0 drawing.cad;
#
quit;
```

Entering (Writing) MQL Commands

An MQL command consists of a keyword and, optionally, clauses and values. MQL commands follow their own set of rules and conventions, as in any other programming or system language. In the following sections, you will learn about the conventions used in this book for entering, as well as reviewing, displayed MQL commands.

You must follow a fixed set of syntax rules so that 3DSpace can interpret the command correctly. If you do not, the system may simply wait until the required information is provided or display an error message.

When in interactive mode, each command is prompted by:

```
mql<#>
```

The syntax rules are summarized in the following table. If you have a question about the interpretation of a command, see the reference section of this guide.

Writing and Entering MQL Commands	
Commands	Consist of words each separated by one or more spaces, tabs, or a single carriage return.
	Begin with a keyword. Most keywords can be abbreviated to three characters (or the least number that will make them unique).
	All NAMES, VALUES, etc. must be enclosed within single or double quotes when they have embedded spaces, tabs, newlines, commas, semi-colons, or pound signs.
	End with either a semicolon (`;`) or double carriage return.
	For example, this is the simplest MQL command. It contains one keyword and ends with a semicolon: `quit;`
Clauses	May or may not be necessary.
	Begin with a keyword.
	Are separated with a space, tab, or a carriage return.

	The following example is a command with two clauses separated by both carriage returns and indented spaces.
	`add attribute “Ship’s Size”`
	`  description “Ship size in metric tons”`
	`  type integer;`

Values	Are separated within clauses by a comma (`,`). Values may be single or, if the keyword accepts a list of values, they can be specified either separately or in a list. For example:
	`attribute Size, Length`
	Or:
	`attribute Size`
	`attribute Length`

Comments	Begin with a pound sign (#) and end with a carriage return. For example: `# list users defined within the database`
	Comments are ignored by 3DSpace and are used for your information only.

MQL Command Conventions

When commands are displayed on your screen, the commands are displayed in diagrams which obey a set of syntax rules. These rules are summarized in the following table:

Reviewing Displayed MQL Commands	
keywords	All keywords are shown in lowercase. These words are required by MQL to correctly process the command or clause. Though they are displayed in all lowercase in the syntax diagram, you can enter them as a mixture of uppercase and lowercase characters.
	All keywords can be abbreviated providing they remain distinctive. For example, you can use `bus` for `businessobject`.

Values	User-defined values are shown in uppercase. The words in the syntax diagram identify the type of value expected. When entering a value, it must obey these rules:
	1. You can enter values in uppercase or lowercase. However, values are case-sensitive. Once you enter the case, later references must match the case you used.
	`For example, if you defined the value as Size, the following values would` *not* `match: SIZE, size, sIZE.`
	2. You must enclose a value in double (`" "`) or single (`' '`) quotes if it includes spaces, tabs, carriage returns (new lines), commas, semicolons (`;`), or pound signs (#).
	A space is included in the following examples, so the values are enclosed in quotes:
	`"Project Size"`
	`'Contract Terms'`
	3. If you need to use an apostrophe within a value, enclose the value in quotes. For example:
	`"Project's Size"`
	4. If you need to use quotes within a value, enclose the value in single quotes. For example:
	`'Contract "A" Terms'`

`[option]`	Optional items are contained within square brackets `[ ]`. You are not required to enter the clause or value if you do not need it.
	Do not include the brackets when you are entering an option.
`{option}`	All items contained within braces `{ }` can appear zero or more times.
	Do not include the braces when you are entering an option.

`|option1|`
`|option2|`
`|option3|`
	All items shown stacked between vertical lines are options of which you must choose one.

Important MQL Commands

When you first use MQL, there are two important MQL commands you should know: Quit and Help.

Quit - exits the current MQL session and returns control to the operating system. To terminate your MQL session, simply enter:

```
quit;
```

When writing the code for program objects, you may want the program to return an integer value to the system that ran the program. For this reason, the quit command can be followed by an integer, as follows:

```
quit [INTEGER];
```

- The INTEGER value is passed and interpreted as the return code.

Help -is available for various MQL command categories. If you do not know which category you want or you want to get a listing of the entire contents of the help file, enter:

```
help all;
```

Eventually, you will probably want help only on a selected MQL category:

application	association	attribute
businessobject	channel	command
config	connection	context
cue	dimension	download
error	export	filter
form	format	group
import	index	inquiry
license	location	menu
monitor	person	policy
portal	program	property
query	relationship	role
rule	set	site
store	table	thread
transaction	type	upload
user	validate	vault

To get help on any of these categories, enter::

```
help MQL_CATEGORY;
```

- help is the keyword.
- MQL_CATEGORY is one of the categories listed above.

If you have a question about the interpretation of a command, see the reference page on that command.

When you enter this command, 3DSpace displays a list of all the commands associated with the category along with the valid syntax. For example, if you enter the following command:

```
help context;
```

It will show the following syntax diagram:

```
set context [ITEM {ITEM}];
 where ITEM is:
    | person PERSON_NAME |
    | password [PASSWORD_VALUE] |
    | vault [VAULT_NAME] |
print context;
```

This information indicates that there are two commands associated with the context category: the set context and print context commands.

The first command has three clauses associated with it: the person, password, and vault clauses. The square brackets ([]) on either side of PASSWORD_VALUE mean that it is optional. If a clause is used, you must obey the syntax of the clause. In the person clause, the syntax indicates that a value for PERSON_NAME is required. Words shown in all uppercase indicate user-defined values. (Refer to Entering (Writing) MQL Commands for a complete description of syntax rules.)

Building an MQL Script

Scripts are useful when you are performing many changes to the 3DSpace database. This is certainly true in the initial building process and it can be true when you are adding a number of old files or blocks of users into an existing database. MQL scripts provide a written record that you can review. Using a text editor makes it easy to duplicate similar blocks of definitions or modifications.

Running Scripts and Using Basic MQL Commands

When building a script, there are several MQL commands that help run other scripts and let you monitor the processing MQL commands.

MQL Commands to Use When Building a Script	
`output VALUE;`	The Output command enables you to print a message to your output device. This is useful within scripts to provide update messages and values while the script is processed. For example, if you are processing large blocks of commands, you may want to precede or end each block with an Output command. This enables you to monitor the progress of the script.
`password PASSWORD;`	The Password command enables you to change the current context password. The Password command enables you to change your own password (which can also be done with the Context command).
`run FILE_NAME [continue];`	The Run command enables you to run a script file. This is useful if you are working in interactive mode and want to run a script. The `continue` keyword allows the script to run without stopping when an error occurs. This is essentially the same as running MQL with the -k option, but it is available at run time, making it usable by programs.
`shell COMMAND;`	The Shell command enables you to execute a command in the operating system. For example, you can perform an action such as automatically sending an announcement to a manager after a set of changes are made. The Output command (see above) sends a message to your output device; but, it cannot send a message to a different device. You can do so with the Shell command.
`verbose [on|off];`	The Verbose command enables you to increase or decrease the amount of message detail generated by the MQL interpreter as the script is processed. This is similar to the -v Option. For more information, see *MQL Command Reference: mql Command* in the online user assistance. When set to `ON`, more detail is provided. When set to `OFF`, only errors and important system messages are displayed.
`version;`	The Version command enables you to see which version of MQL you are using. For example, this is useful if you want a record of the version used to process a script.

Using Comments

Comments visually divide scripts into manageable sections and remind you of why structures and objects were defined and modified. Comments, which are preceded in the script by a pound sign (#), are ignored by MQL:

```
# script comments
```

Each comment line must begin with a pound sign (#).

Parameterized MQL Commands

Using parameterized MQL prevents MQL injection flaws in apps. An MQL injection attack consists of insertion or “injection” of an MQL command via the input data from the client to the application. A successful MQL injection exploit can read sensitive data from the database, modify database data (e.g., through Insert/Update/Delete commands), execute administration operations on the system (e.g., Clear Vault command), and in some cases, issue commands to the operating system.

The following Java example is unsafe because it allows an attacker to inject code into the command that the system would execute:

```
String command = "temp query bus Part * * where owner == " +
    request.getParameter("customerName");
mql.executeCommand(context, command);
```

If `"customerName"` is equal to `"; clear all"`, then the `"temp query"` will fail because the command is incomplete, but the `"clear all"` command will execute as a valid command.

MQL injection flaws are introduced when software developers create dynamic commands that include user-supplied input. Avoiding MQL injection flaws is straightforward. Developers should either stop writing dynamic commands, or prevent user-supplied input that contains malicious MQL from affecting the logic of the executed command.

In support of this, the Open Web Application Security Project (OWASP) recommends that software developers apply the following defenses to prevent MQL injection attacks:

- Use prepared statements (parameterized commands).
- Use stored procedures.
- Escape all user-supplied input.
- Enforce "least privilege" (e.g., by minimizing the privilege of user accounts). See *Least Privilege*, below.
- Perform "white list input validation" (e.g., by providing a list of the available inputs, such as attribute ranges). See *White List Input Validation*, below.

The Server already provided capabilities to implement stored procedures (Java Program Objects or JPOs), enforce "least privilege" (P&O security) and to perform "white list input validation" (attribute ranges). Additionally, web application development teams use APIs that allow them to encode and escape data.

You can use parameterized MQL commands to first to define the MQL command and then pass in each parameter to the command at a later time. This coding style makes it possible to distinguish between code and data, regardless of what user data is supplied.

Escaping of user-supplied input is unnecessary in MQL. In other words, for code that read:

```
"temp query bus '"+type+"' Part * select id"
```

where `"type"` was bracketed by two single quotes, now rather than writing:

```
"temp query bus '$1' Part * select id"
```

you can simply write:

```
"temp query bus $1 Part * select id"
```

Parameterized commands ensure that an attacker is not able to change the intent of a query, even if the MQL commands are inserted by the attacker. In the following safe example, if an attacker were

to enter a userID of `tom' or '1'='1`, the parameterized query would not be vulnerable and would instead look for a username that literally matches the entire string `tom' or '1'='1`.

```
String customerName = request.getParameter("customerName");
String whereClause = "'owner == " + customerName + "'";
String command = "temp query bus $1 $2 $3 where $4";
mql.executeCommand(context, command, "Part", "*", "*",
  whereClause);
```

For details on how to implement parameterized MQL commands, see the Javadoc online help for the matrix::db::MQLCommand::executeCommand() method.

When passing a multi-value argument, you must pass in multiple parameters for that argument. For example, for an MQL command that contains:

```
...value1,value2,value3...
```

the MQLCommand.executeCommand must define:

Format: `...$n,$n+1,$n+2,...`
Values: `"value1", "value2", "value3", …`

For example, this command defines a multi-value attribute:

```
mql.executeCommand(ctx, "mod bus $1 $2 $3,$4", boID1, "str_att", "str1", "str2");
```

This command defines multi-value access:

```
mql.executeCommand(ctx, "mod bus $1 add access bus $2 for $3 as $4,$5,$6",
boID1, boID2, "comment", "read", "show", "checkout");
```

Building an Initial Database

Building a database usually involves writing two scripts. One script will contain all the definitions used by the database. The second script creates business objects, associates files with the objects, and defines relationships between objects.

By separating these two steps, it is easier to test the database. You cannot create business objects or manipulate them until the definitions are in place. By first creating and testing all the definitions, you can see if the database is ready to add objects. You may want to alter the definitions based on test objects and then use the altered definitions for processing the bulk of the object-related data.

Clearing the Database

When you are creating an initial database and you want to start over, the Clear All command enables you to work with a clean slate. The Clear All command clears all existing definitions, vaults, and objects from the database. It wipes out the entire content of the database and leaves only the shell. While this is useful in building new databases, it should NOT be used on an existing database that is in use.

Once this command is processed, the database is destroyed and can only be restored from backups.

Only the person named “creator” can use the Clear All command.

```
clear all;
```

Do not attempt to add the persons creator, guest, or Test Everything using MQL. Adding or modifying these objects could cause triggers, programs or other application functions not to work.

The clear all command should NEVER be used while users are on-line.

Clearing Vaults

To test and build new vaults, use the Clear Vault command. This command ensures that the vault does not contain any business objects. It can be run by a person with System Administrator privileges.

```
clear vault VAULT_NAME;
```

- VAULT_NAME is the names of the vault(s) to be cleared.

When the Clear Vault command is processed, all business objects are cleared within the named vaults. The definitions associated with the vault and the database remain intact. Only the business objects within the named vaults are affected.

The clear vault should NEVER be used while users are on-line.

Creating Definitions in a Specific Order

When creating the script of definitions for the initial database, the order in which you make the definitions is significant. Some commands are dependent on previous definitions. For example, you cannot assign a person to a group if that group does not already exist. For this reason, it is recommended that you write your definition commands in the following order

:

Definition Order		
1	Roles Groups Persons Associations *Or:* Persons Groups Roles Associations	You can define roles, groups, and persons either way depending on your application. Since associations are combinations of groups and roles, they must be created after groups and roles are defined.
2	Attributes Types Relationships Formats Stores Policies	Order is important for this set of definitions. For example, types use attributes, so attributes must be defined before types.
3	Vaults	Vaults can be defined at any time although they are most commonly defined last.

Processing the Initial Script

After your script is written, you can process it using the mql command, described in *Accessing MQL*. As it is processed, watch for error messages and make note of any errors that need correction.

Once the script that creates the definitions is successfully processed, you can use either the interactive MQL interpreter or the Business Administrator account to check the definitions. From Matrix Navigator or MQL, you can create objects, associate files, and otherwise try out the definitions you have created. If you are satisfied with the definitions, you are ready to write your second MQL script.

Writing the Second MQL Script

The second MQL script in this example creates business objects, manipulates them, and defines relationships. For more information on the types of information commonly found in this second script file, see *Manipulating Data* in Chapter 3.

Modifying an Existing Database

After you define the initial database, you may want to modify it to add additional users and business objects. If the number of users or the amount of information is large, you should consider writing a MQL script. This enables you to use a text editor for editing convenience and provides a written record of all of the MQL commands that were processed.

Often when you are adding new business objects, you will define new vaults to contain them. This modular approach makes it easier to build and test. While some new definitions may be required, it is possible that you will use many of the existing definitions. This usually means that the bulk of the modification script will involve defining objects, manipulating them, and assigning relationships. These actions are done within the context of the vault in which the objects are placed.

To test and build new vaults, use the Clear Vault command. This command ensures that the vault does not contain any business objects. See *Clearing the Database* for details.

When writing your script for modifying an existing database, remember to include comments, occasional output commands, and a Quit command at the end (unless you'll want to use the interactive MQL interpreter after script completion).

Working with Transactions

A *transaction* involves accessing the database or producing a change in the database. All MQL commands involve transactions.

Implicit Transactions

When you enter an MQL command, the transaction is started and, if the command is valid, the transaction is *committed* (completed). For example, assume you enter the following MQL command:

```
add person Debbie;
```

As soon as you enter the command, the system starts the transaction to define a person named Debbie. If that person is already defined, the command is invalid and the transaction aborts—the command is not processed and the database remains unchanged. If the person is not already defined, the app is committed to (the completion of) the transaction to add a new person. Once a transaction is committed, the command is fully processed and it cannot be undone. In this case, Debbie would be added to the database.

Ordinarily, starting, committing, and aborting transactions is handled *implicitly* with every MQL command. Each command has two implied boundaries: the starting keyword at the beginning of the command and the semicolon or double carriage return at the end. When you enter an MQL command, a transaction is implicitly started at the keyword and is either committed or aborted depending on whether the content of the command is valid.

This implicit transaction control can also be performed explicitly.

Explicit Transaction Control

Several MQL commands enable you to explicitly start, abort, and commit transactions:

`abort transaction [NAME];`
`commit transaction;`
`print transaction;`
`start transaction [read];`
`set transaction wait|nowait|savepoint [NAME];`

These commands enable you to extend the transaction boundaries to include more than one MQL command.

- If you are starting a transaction, use the `read` option if you are only reading.
- Without any argument, the `start transaction` command allows reading and modifying the database.

Extending Transaction Boundaries to Include Several Commands

Including several MQL commands within a single set of transaction boundaries enables you to tie the success of one command to the success of some or all of the other MQL commands.

It will appear that all valid MQL commands are performed; but, they are not permanent until they are committed. You should not quit until you terminate the transaction or you will lose all your changes. Also, the longer you wait before committing changes, the more likely you are to encounter

an error message, particularly if you are entering the commands interactively when typing errors are common.

Let's look at an example in which you want to create a set of business objects and then associate a collection of files with those objects. You might successfully create the objects and then discover that you cannot place the files in them. With normal script or interactive MQL processing, the objects are created even though the checkin fails. By altering the transaction boundaries, you can tie the successful processing of the files to the creation of the business objects to contain them. For example:

```
start transaction update;
add businessobject Drawing "C234 A3" 0
   policy Drawing
   description "Drawing of motor for Vehicle V7";
checkin businessobject Drawing "C234 A3" 0 V7-234.MTR;
add businessobject Drawing "C234a A3" 0
   policy Drawing
   description "Drawing of alt. motor for Vehicle V7";
checkin businessobject Drawing "C234a A3" 0 V7-234a.MTR;
commit transaction;
```

This transaction is started and the current state of the database is saved. The `add businessobject` commands create business objects, and the `checkin businessobject` commands add the files.

When the `commit transaction` command is processed, MQL examines all the commands that it processed since the `start transaction` command. If no error messages are detected, all changes made by the commands are permanently made to the database.

If ANY errors are detected, NONE of the changes are committed. Instead, the system returns the database to the recorded state it was in at the start of the transaction. This essentially gives you an *all or nothing* situation in the processing of MQL commands. However, savepoints can be set in the transaction to be used in conjunction with the `abort transaction` command, which can save some of the work already done.

The `set transaction savepoint [NAME]` command allows you to set a point within a transaction that will be used for any required rollbacks. You cannot set a transaction savepoint when you have an open Iterator.

If you specify the `NAME` in an `abort transaction` command, the transaction is rolled back to the state where the savepoint was created. If no name is specified, the entire transaction is rolled back. Of course the transaction must still be committed before the commands between the `start transaction` and the savepoint are processed. The `set transaction savepoint NAME` command can be issued multiple times without error. The effect is that the savepoint is changed from the original savepoint state in the transaction to the current state of the transaction. The `abort transaction NAME` command can also be issued multiple times.

If `set transaction savepoint NAME` is issued and the transaction has already been marked as aborting, the command will fail and return an error. When using this command via Studio Customization Toolkit or Tcl, be sure to check for the returned error "Warning: #1500023: Transaction aborted". If you assume `set transaction savepoint NAME` works, a subsequent `abort transaction NAME` may roll the transaction back to an older state than expected.

The `wait` and `nowait` arguments are used to tell the system if you want to queue up for locked objects or not. When `nowait` is used, an error is generated if a requested object is in use. The default is to wait; the `wait` keyword is a toggle.

The Oracle SQL `nowait` function is used by Oracle only.

One advantage to using the transaction commands to process command blocks involves the processing time. Commands processed within extended transaction boundaries are processed more quickly than the normal command transactions.

If you choose to process blocks of MQL commands within explicit transaction boundaries, you need to carefully consider the size and scope of the commands within that transaction. When you access anything with an MQL command, that resource is locked until the transaction is completed. Therefore, the more commands you include within the transaction boundaries, the more resources that will be locked. Since a locked resource cannot be accessed by any other user, this can create problems if the resource is locked at a time when heavy use is normal. The use of large transactions may also require you to adjust the size of your Oracle rollback segments. See your Oracle documentation for more details.

Be careful that the blocks are not too large.

You should immediately attend to an explicit transaction that either has errored or is awaiting a `commit` or `abort` command. Many database resources other than objects are also locked with the pending transactions. Users will begin to experience lock timeouts as they attempt typical database operations.

Access Changes Within Transactions

A variety of information about access rights are cached during a transaction within the 3DExperience platform. This information includes not only the security model, but also other parts of the administration model, such as which attributes belong to which types. These caches are not reset until the transaction is committed, which guarantees the integrity of all updates within a transaction.

Changes to the metamodel - that is, any part of the administration model that influences the behavior of the system - must be done within their own transactions. The metamodel includes all administrative types, but can also include data on business objects or connections that control behavior, such as when an attribute value is referenced in an access rule. This type of data is cached as part of the metamodel, and therefore if you make changes to them within a longer transaction, you may get unexpected results.

Specifically, if the current context gains access to an object within a transaction due to an access expression that evaluates to TRUE based on a current attribute value, then that access is granted for the entire transaction, even if the attribute value is changed. The same could be said about changing the user's group or role assignments. With regard to group and role assignments, if you have access at the beginning of a transaction, you will have it at the end. That is, once a transaction is begun, the system does not check back into the database to detect changes to person assignments that may affect the current user's access within the person's cache.

Consider the example below:

```
set context user creator;
add group GRP1;
add person PERS1 type application,full;
add type ACCESS attribute int-u;
add policy NONE type ACCESS state one public none owner all user GRP1
read,show,modify;
add bus ACCESS a1 0 vault unit1 policy NONE int-u 1;
# First, get the object in PERS1's cache by failing to modify attr:
```

```
start trans;
set context user PERS1;
mod bus ACCESS a1 0 int-u 2;
# Now switch back to creator and modify the group so that PERS1 can
modify
# >>>>> OR make this change from a separate MQL/Business session.
set context user creator;
mod person PERS1 assign group GRP1;
# CASE 1: Try to modify - this is not allowed
set context user PERS1;
mod bus ACCESS a1 0 int-u 2;
commit trans;
# Now switch back to creator and modify object to kick it out of cache
set context user creator;
mod bus ACCESS a1 0 int-u 3;
# Modify the group so that PERS1 can modify
mod person PERS1 assign group GRP1;
# Start a transaction and get the object in PERS1's cache with
successful mod
start trans;
set context user PERS1;
mod bus ACCESS a1 0 int-u 4;

# Now switch back to creator and modify the group so that PERS1 can NOT
modify
# >>>>> OR make this change from a separate MQL/Business session.
set context user creator;
mod person PERS1 remove assign group GRP1;

# CASE: Try to modify - it is allowed.
set context user PERS1;
mod bus ACCESS a1 0 int-u 5;
commit trans;
```

Transaction Triggers

You can configure a transaction trigger that will be invoked one time for all events on a type or relationship at the end of a transaction. When defined on a type or relationship, whenever a business object or connection of that type or relationship is modified, at the end of the outside transaction, the program associated with the transaction trigger is invoked.

If both a transaction trigger and a general modify trigger are defined for a type, both triggers will be invoked at the end of the outside transaction. Transaction triggers can only be Action triggers; and cannot be configured as a Check or Override trigger.

The programs associated with transaction triggers must be JPOs. In addition, these programs will always run as deferred. If configured to run immediately, that setting is ignored and program execution is still deferred. Because the program is deferred, if any errors occur during the program execution, all prior changes within the transaction boundary are not rolled back. In addition, the trigger program should not trigger another transaction event on the same business object or connection; if it does, an error will be thrown.

Transaction trigger JPOs are not invoked for objects that are deleted inside a transaction.

Macros specific to the type or relationship are not available to the trigger program. Only one macro can be passed as an input parameter to a program for a transaction trigger:

`$TRANSHISTORY`

`$TRANSHISTORY` contains history items for operations in the current transaction, and is a carriage return delimited string listing all changes to the business object within the transaction. Programs should process the output of this macro in the same manner as output from `print bus T N R select history`. The string size limit for this macro is 2^31-1 bytes. If this limit is exceeded, the program call is split and multiple calls may be made depending on the amount of history information.

Each time a transaction trigger is invoked, information on all events that have occurred is passed to the program. For example, if business object 1 is promoted and the owner of business object 2 is changed, at the end of the transaction, the transaction trigger is invoked on time and `$TRANSHISTORY` is populated.

History information is passed to the trigger program even if history is turned off.

The following business object events support transaction triggers:

addinterface	connect	modifyattribute
changename	copy	modifydescription
changeowner	create	removefile
changepolicy	delete	removeinterface
changetype	disconnect	revision
changevault	grant	revoke
checkin	lock	unlock
checkout		

The following connection events support transaction triggers:

create	addinterface	modifyto
delete	modifyattribute	removeinterface
freeze	modifyfrom	thaw

Working With Threads

MQL has a notion of a “thread,” where you can run multiple MQL sessions inside a single process. Each thread is like a separate MQL session, except that all threads run within the same MQL process. This concept was introduced into MQL to support the MSM idea of multiple “deferred commit” windows.

For more information, see *MQL Command Reference: thread Command* in the online user assistance.

Using Tcl

To enter Tcl mode, enter the following in MQL:

```
tcl;
```

In Tcl mode, the default Tcl prompt is % and MQL will accept only Tcl commands. Native MQL commands are accessed from Tcl by prefixing the command with the keyword mql. For example:

```
% mql print context;
```

Otherwise, Tcl command syntax must be followed. It differs from MQL in several ways:

Tcl Command Syntax Differences	
Comments	If the first nonblank character is #, all the characters up to the next new line are treated as a comment and discarded. The following is not a comment in Tcl: `set b 101; #this is not a comment`
Command Separators	New lines are treated as command separators. They must be preceded by a backslash (\) to not be treated as such. A semi-colon (;) also is a command separator.
Commas	For MQL to properly parse commands with commas, the commas must be separated with blanks. For example: `access read , write , lock`
Environment Variables	Environment variables are accessible through the array, `env`. For example, to access the environment variable `MATRIXHOME`, use: `env(MATRIXHOME)`
History	Tcl has its own history mechanism.

For a description of common Tcl syntax errors, see *Tcl Syntax Troubleshooting*.

Command line editing is not currently available.

The Tcl command, exit, returns you to native MQL. Like the MQL Quit command, Exit can be followed by an integer value which is passed and interpreted as the return code of the program.

Displaying Special Characters

When displaying special symbols in a Tk window, it may be necessary to specify the font to be used. Tcl/Tk 8.0 tries to map any specified fonts to existing fonts based on family, size, weight, etc. Depending on where you are trying to use a special symbol, you may be able to use the -font arg.

MQL and Tcl

When Tcl passes an MQL command to MQL, it first breaks the command into separate arguments. It breaks the command at every space, so you have to be careful to use double quotes ("") or braces ({}) to enclose any clause which is to be passed along to the system as a single argument.

The parsing of Tcl commands containing backslashes within an MQL program object is different than when those same commands are run from MQL command prompt.

For example, create a text file with the command lines:

```
tcl;
set a "one\\two\\three"
puts "a contains $a"
puts "b will use slashes instead of backslashes"
regsub -all {\\} $a {/} b
puts "now b contains $b"
puts "now b contains $b"
```

Run this file from MQL (or alternatively type each line in MQL command window). The final line of output will be:

```
now b contains one/two/three
```

This is consistent with how the same lines of code behave in native Tcl.

However if you use this same code as a MQL program object, to get the same output you need to use triple rather than double backslashes so your code becomes:

```
tcl;
set a "one\\\two\\\three"
puts "a contains $a"
puts "b will use slashes instead of backslashes"
regsub -all {\\\} $a {/} b
```

Tcl Syntax Troubleshooting

The Tcl executable has strict rules regarding syntax, which can cause problems when the rules are not followed. Below are some tips for avoiding common syntax-related errors.

1. Extra Spaces

A common error is to include extra spaces in a line of Tcl. You can easily miss excess white space, but the Tcl compiler examines white space closely. This can happen when you join two lines together during editing and something is deleted. An extra space before the carriage return may end up on the line of code. Tcl will catch this error every time, but it won't report anything back to you. A quick way to catch this error is to write the program to a text file and to search for all occurrences of a blank space. The syntax for this is:

```
grep -n "{ $" *.<text_file_name_suffix>
```

The second common error occurs with extra spaces in an attribute name in an MQL select command. For example:

```
set sClass [ mql print businessobject $sOid select
attribute\[Classification \] dump $cDelimiter ]
```

In using the Classification attribute, you may unintentionally leave an extra space before the "\" character. The correct form of this query looks like this:

```
set sClass [ mql print businessobject $sOid select
attribute\[Classification\] dump $cDelimiter ]
```

To check for this type of problem, first write all the programs to text files, then search the files for spaces prior to a backslash. The syntax for searching this is:

```
grep -n "attribute\\\\\[" *.tcl | grep " \\\\\]"
```

2. Matching Braces

Another common error results when the Tcl compiler determines that the number of begin/end braces does not match. The typical cause for this error is commented-out code containing braces, which the Tcl compiler includes in its brace count, leading to unexpected results. For example, the

code sequence below looks like it should run correctly. However, the Tcl compiler will determine that the brace count does not match, causing problems.

```
# foreach sUser $lUserList {
foreach sUser $lEmployeeList {
   puts $sUser
}
```

The correct solution is to either comment-out the entire block of code, or to delete it, so the compiler will generate a correct brace match, for example:

```
# foreach sUser $lUser {
#     puts $sUser
# }

foreach sUser $lEmployeeList {
   puts $sUser
}
```

or

```
foreach sUser $lEmployeeList {
   puts $sUser
}
```

Again, write the program to a text file, search for the braces, and ensure that all braces match. The syntax for retrieving the lines containing braces and storing them in a file is:

```
grep "\#" *.tcl | grep "\{" | cut -d: -f1,1 > filelist.log
```

The file filelist.log will then contain only the lines with braces for analysis.

3. Tcl Versions

To determine what version of Tcl you are using, switch to tcl mode then use the info tclversion or info patchlevel commands:

```
MQL<1>tcl;
% info tclversion
8.3.3
```

Or

```
% info patchlevel
8.3.3
```

The Tk library must be loaded in order to retrieve the Tk version. On Windows, a command similar to the following should be used:

```
% load tk83
% info loaded
{tk83 Tk}
```

For More Information

For more information about Tcl and Tk, refer also to the following books:

Ousterhout, John K., Tcl and the Tk Toolkit. Addison-Wesley, April 1994, ISBN 0-201-63337-X.

Welch, Brent, Practical Programming in Tcl and Tk. Prentice Hall, May 1995, ISBN 0-13-182007-9.

World Wide Web documents for reference:

USENET newsgroup: comp.lang.tcl

Parameterized MQL Commands

Using parameterized MQL prevents MQL injection flaws in apps. An MQL injection attack consists of insertion or "injection" of an MQL command via the input data from the client to the application. A successful MQL injection exploit can read sensitive data from the database, modify database data (e.g., through Insert/Update/Delete commands), execute administration operations on the system (e.g., Clear Vault command), and in some cases, issue commands to the operating system.

The following Java example is unsafe because it allows an attacker to inject code into the command that the system would execute:

```
String command = "temp query bus Part * * where owner == " +
    request.getParameter("customerName");
mql.executeCommand(context, command);
```

If `"customerName"` is equal to `"; clear all"`, then the `"temp query"` will fail because the command is incomplete, but the `"clear all"` command will execute as a valid command.

MQL injection flaws are introduced when software developers create dynamic commands that include user-supplied input. Avoiding MQL injection flaws is straightforward. Developers should either stop writing dynamic commands, or prevent user-supplied input that contains malicious MQL from affecting the logic of the executed command.

In support of this, the Open Web Application Security Project (OWASP) recommends that software developers apply the following defenses to prevent MQL injection attacks:

- Use prepared statements (parameterized commands).
- Use stored procedures.
- Escape all user-supplied input.
- Enforce "least privilege" (e.g., by minimizing the privilege of user accounts). See *Least Privilege*, below.
- Perform "white list input validation" (e.g., by providing a list of the available inputs, such as attribute ranges). See *White List Input Validation*, below.

The Server already provided capabilities to implement stored procedures (Java Program Objects or JPOs), enforce "least privilege" (P&O security) and to perform "white list input validation" (attribute ranges). Additionally, web application development teams use APIs that allow them to encode and escape data. Release V6R2013 adds the ability to use parameterized MQL commands. Parameterized MQL commands allow the developer first to define the MQL command and then to pass in each parameter to the command at a later time. This coding style makes it possible to distinguish between code and data, regardless of what user data is supplied.

Parameterized commands ensure that an attacker is not able to change the intent of a query, even if the MQL commands are inserted by the attacker. In the following safe example, if an attacker were to enter a userID of `tom' or '1'='1`, the parameterized query would not be vulnerable and would instead look for a username that literally matches the entire string `tom' or '1'='1`.

```
String customerName = request.getParameter("customerName");
String whereClause = "'owner == " + customerName + "'";
String command = "temp query bus $1 $2 $3 where $4";
mql.executeCommand(context, command, "Part", "*", "*",
  whereClause);
```

For details on how to implement parameterized MQL commands, see the Javadoc online help for the matrix::db::MQLCommand::executeCommand() method.

Least Privilege

To minimize the potential damage of a successful MQL injection attack, you can minimize the privileges assigned to every account in the system. Do not assign "business" or "system" access rights to standard application accounts. In many cases, this is easy and everything just works when done this way, but it is very dangerous. It is preferable to start from the ground up to determine what access rights the application accounts require, rather than trying to figure out what access rights you need to take away. Make sure that accounts that only need read access are only granted read access and only to the data to which they need to access.

The privileges of the operating system account for the Java server for MCS and FCS should be minimized. Do not run the Java server application as `root`! Create an OS account to something more appropriate and with restricted privileges.

The system provides extensive capabilities to manage access to data. This access control can be applied at the group, role, or individual user level. It can also be applied against a class of data types, or as granularly as individual attributes.

White List Input Validation

White list input validation is appropriate for all input fields in which the user enters data. It involves defining exactly what data is authorized; by definition, everything else is not authorized. If the data is well-structured (such as dates, social security numbers, zip codes, email addresses, and the like), then the developer should be able to define a very strong validation pattern, usually based on regular expressions, for validating such input. If the input field comes from a fixed set of options, such as a drop-down list or radio buttons, then the input can only match exactly one of the values offered in the first place.

The most difficult fields to validate are free-text fields, such as blog entries. However, even those types of fields can be validated to some degree, for example by excluding all non-printable characters and defining a maximum size for the input field.

The system provides capabilities to create attribute ranges and to perform input validation in web user interfaces.

2 Working with Metadata

This chapter explains the metadata available in the 3DEXPERIENCE Platform.

In this section:

Attributes

An *attribute* is any characteristic that can be assigned to an object or relationship. Objects can have attributes such as size, shape, weight, color, materials, age, texture, and so on.

You must be a Business Administrator to add or modify attributes. (Refer also to the Business Modeler Guide.)

Defining an Attribute

Before types and relationships can be created, the attributes they contain must be defined as follows:

```
add attribute NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the attribute.

`ADD_ITEM` is an Add Attribute clause that provides additional information about the attribute you are creating.

For more information, see *MQL Command Reference: attribute Command* in the online user assistance.

Assigning Attributes to Objects

In the 3DEXPERIENCE Platform, objects are referred to by type, name, and revision. Object types are defined by the attributes they contain. When an object is created, you specify the object type and then 3DSpace prompts for attribute values for that object instance.

For example, assume the object is clothing. It might have attributes such as article type (for example, pants, coat, dress, shirt, and so on), size, cost, color, fabric type, and washing instructions. Now assume you are creating a new article of clothing. When you create this object, 3DSpace prompts you to provide values for each of the assigned attributes. You might assign values such as jacket, size 10, $50, blue, wool, and dry clean.

The specific value for each attribute can be different for each object instance. However, all objects of the same type have the same attributes.

Assigning Attributes to Relationships

Like business objects, a relationship may or may not have attributes. Since a relationship defines a connection between two objects, it has attributes when there are characteristics that describe that connection.

For example, an assembly could be connected to its components with a relationship called, “component of,” which could have an attribute called, “quantity.” When the component and the assembly are connected, the user would be prompted for the quantity or number of times the component is used in the assembly.

The same attributes could apply to either the objects or the relationship. When an object requires additional values for the same attribute in different circumstances, it is easier to assign the attributes to the relationship. Also, determine whether the information has more meaning to users when it is associated with the objects or the relationship.

Assigning Attribute Types

The Type clause is always required. It identifies the type of values the attribute will have. An attribute can assume one of these types of values: binary, string, Boolean, real, integer, date, shortbinary, and enum. Shortbinary attributes contain internal IDs and enum attributes contain range values. When determining the attribute type, you can narrow the choices by deciding if the value is a number or a character string. Once a type is assigned to an attribute, it cannot be changed. The user can associate only values of that type with the attribute.

Assigning Attribute Ranges

You can define a range of values the attribute can assume. This range provides a level of error detection and gives the user a way of searching a list of attribute values. If you define an attribute as having a specific range, any value the user tries to assign to that attribute is checked to determine if it is within that range. Only values within all defined ranges are allowed. There are several ways to define a range:

- Using a relational operator, such as is less than (“<”) or is equal to (“=”)
- Using a range with a pattern, or special character string, such as starts with A (“A*”)
- Using multiple ranges to provide a list of allowed values
- Using a program to define a range depending on conditions

See also *Multi-Value and Range-Value Attributes*.

Assigning a Default Value

When a business object is created and the user does not fill in the attribute field, the default value is assigned. When assigning a default value, the value you give must agree with the attribute type. If the attribute should contain an integer value, you should not assign a string value as the default.

For example, assume you want to define an attribute called PAPER_LENGTH. Since this attribute will specify the size of a sheet of paper, you defined the type as an integer. For the default value, you might specify 11 inches or 14 inches, depending on whether standard or legal size paper is more commonly used.

Applying a Dimension to an Existing Attribute

You can add a dimension to attributes of type integer or real, whether or not the attributes already belong to instantiated business objects. The existing value stored for that attribute will be treated as the normalized value.

However, if the existing value is a unit other than what the dimension uses as its default, you need to convert those values. For example, an existing attribute Length has stored values where the business process required the length in inches. You want to add a Length dimension to the Length attribute, and that dimension uses centimeters as the default. If you just add the dimension to the attribute, the existing values will be considered as centimeters without any conversions.

As another example, the Length attribute can be used with multiple types. Some types may use the attribute to store lengths in millimeters, while others store lengths in meters. Because there is only one attribute Length that represents two kinds of data, the dimension cannot be applied without normalizing the business types on the same unit.

Use the convert attribute command to convert the attribute to 1 unit of measure.

```
mod attribute ATTRIBUTE_NAME dimension DIMENSION_NAME;
```

Replace ATTRIBUTE_NAME with the attribute name and DIMENSION_NAME with the dimension name.

Specify the Existing Units of the Attribute

```
convert attribute ATTRIBUTE_NAME to unit UNIT_NAME on temp query bus
“TYPE” * *;
```

This command applies the unit UNIT_NAME to the attribute, which causes the conversion from the existing value with the applied units to normalized values.

The system displays this message:

```
Convert commands will perform mass edits of attribute values.
You should consult with MatrixOne support to ensure you follow
appropriate procedures for using this command.
Proceed with convert (Y/N)?
```

If the system message includes the following warning,

```
WARNING: The search criteria given in the convert command includes objects
that currently have unit data. This convert command will overwrite that
data.
```

the data for this attribute has already been converted and you should not continue (type N).

Changing the Default (Normalized) Units of a Dimension

The Weight attribute included in Business Process Services, is defined to have a dimension containing units of measure with the default units set to grams. If your company uses the Weight attribute representing a different unit, you have these options:

- Convert existing values to use the new dimension with its default as described in *Applying a Dimension to an Existing Attribute*.
- Change the Dimension’s default units, offsets, and multipliers as described in this section

After applying a dimension, you can only change the default units of measure before any user enters any data for the attribute. After values are stored in the as entered field, you cannot change the default units of measure.

To change a dimension’s default units

1. Using Business Modeler, locate the dimension and open it for editing.
2. Click the **Units** tab.
3. For the unit you want to use as the default:
   - **a)** Highlight the Unit.
   - **b)** Change the multiplier to 1.
   - **c)** Change the offset to 0.
   - **d)** Check the **Default** check box.
   - **e)** Click **Edit** in the Units section.
4. For all other units:
   - **a)** Highlight the Unit.

b) Change the multiplier to the appropriate value.

c) Change the offset to the appropriate value.

d) Click **Edit** in the Units section.

5. Click **Edit** at the bottom of the dialog box.

Multiple Local Attributes

It is possible to manage local attributes with the same name defined on multiple interfaces on the same object/relationship instance. The objects/relationships can also have a global or local attribute with the same attribute name.

You can add an interface to a business object if the business object hierarchy had a local attribute with the same name as one of the interface attributes as shown in the following MQL session. The naming convention for the local attributes, `I1.a1,` denotes the local attribute `a1` belonging to interface `I1`.

```
MQL<35>add type T1;
MQL<36>add attribute a1 type string default a1-def-value;
MQL<37>mod type T1 add attribute a1;
MQL<38>add interface I1 type T1;
MQL<39>add attribute a1 type string owner interface I1 default
I1-a1-def-value;
MQL<40>add bus T1 test 0 vault unit1 policy simple-u;
MQL<41>mod bus T1 test 0 add interface I1;
MQL<42>print bus T1 test 0 select attribute;
business object T1 test 0
    attribute = a1
    attribute = I1.a1
MQL<43>print bus T1 test 0 select attribute.value;
business object T1 test 0
    attribute[a1].value = a1-def-value
    attribute[a1].value = I1-a1-def-value
MQL<44>print bus T1 test 0 select attribute[a1];
business object T1 test 0
    attribute[a1] = I1-a1-def-value
    attribute[a1] = a1-def-value
MQL<45>print bus T1 test 0 select attribute[*a1];
business object T1 test 0
    attribute[a1] = I1-a1-def-value
    attribute[a1] = a1-def-value
MQL<46>print bus T1 test 0 select attribute[I1.a1];
business object T1 test 0
    attribute[a1] = I1-a1-def-value
```

You can add two interfaces on a business object if both interfaces owned a local attribute with the same name, as shown in the following MQL trace:

```
MQL<28>add type T1;
MQL<29>add interface I1 type T1;
MQL<30>add interface I2 type T1;
MQL<31>add attribute a1 type string owner interface I1 default
I1-a1-def-value;
```

```
MQL<32>add attribute a1 type string owner interface I2 default
I2-a1-def-value;
MQL<33>add bus T1 test 0 policy simple-u vault unit1;
MQL<34>mod bus T1 test 0 add interface I1;
MQL<35>mod bus T1 test 0 add interface I2;
MQL<36>print bus T1 test 0 select attribute;
business object T1 test 0
    attribute = I1.a1
    attribute = I2.a1
MQL<37>print bus T1 test 0 select attribute.value;
business object T1 test 0
    attribute[a1].value = I1-a1-def-value
    attribute[a1].value = I2-a1-def-value
MQL<38>print bus T1 test 0 select attribute[a1];
business object T1 test 0
    attribute[a1] = I2-a1-def-value
    attribute[a1] = I1-a1-def-value
MQL<39>print bus T1 test 0 select attribute[*a1];
business object T1 test 0
    attribute[a1] = I2-a1-def-value
    attribute[a1] = I1-a1-def-value
MQL<40>print bus T1 test 0 select attribute[I1.a1];
business object T1 test 0
    attribute[a1] = I1-a1-def-value
MQL<41>print bus T1 test 0 select attribute[I2.a1];
business object T1 test 0
    attribute[a1] = I2-a1-def-value
```

Attribute Names

A fully qualified name includes both the object name and the attribute name, using the format *ADMIN.ATTRIBUTE*. An unqualified name includes the attribute name only. The only time the fully qualified name of an attribute is used is when `select attribute` is specified. Otherwise, the display name is always the unqualified attribute name. This enables multiple attributes to have the same name, because if the names are displayed "fully qualified," (i.e., `T1.a1`), they do not appear as having the same name.

No matter how the selectable is written (`attribute[X]` where `x` = `I1.a1`, `I2.a1`, `a1`, etc.), the returned attribute is always the simple/common attribute name. The correct value is always returned if you select using the fully qualified local attribute name (*ADMIN.ATTRIBUTE*, for example, `T1.a1`). For example:

```
MQL<219>print bus tp1 test2 1 select attribute[i1.*];
business object tp1 test2 1
    attribute[i1.a1] = 3DPLM
MQL<220>print bus tp1 test2 1 select attribute[].value;
business object tp1 test2 1
    attribute[tp1.a1].value = 3DPLM
    attribute[a1].value = geo
    attribute[i1.a1].value = 3DPLM
MQL<221>print bus tp1 test2 1 select attribute[a1];
business object tp1 test2 1
```

```
    attribute[tp1.a1] = 3DPLM
    attribute[i1.a1] = 3DPLM
    attribute[a1] = geo
```

One limitation of this feature is that there is no way to select only local attributes. For example, specifying `attribute[*.a1]` returns an error:

```
MQL<55>print bus T1 test 0 select attribute[*.a1];
Error: #1900068: print business object failed
Error: #1500063: Unknown field name 'attribute[*.a1]'
```

However, you can use '*' to match all global and local attributes matching a pattern, for example:

```
MQL<4>add type k1;
MQL<5>add interface k1 type k1;
MQL<6>add attribute ak1 type string owner interface k1;
MQL<7>add interface k2 type k1;
MQL<8>add attribute ak1 type string owner interface k2;
MQL<9>add attribute ak1 type string;
MQL<10>mod type k1 add attribute ak1;
MQL<11>add bus k1 test 0 vault unit1 policy simple-u;
MQL<12>mod bus k1 test 0 add interface k1 add interface k1 add
interface k2;
MQL<13>print bus k1 test 0 select attribute [*k1];
business object k1 test 0
   attribute[k2.ak1] =
   attribute[ak1] =
   attribute[k1.ak1] =
```

Using Patterns for Attribute Names in Temp Query Bus Where clause

When a pattern is provided for an attribute name in a `temp query bus' where` clause, the pattern is resolved against all attributes, not just attributes available for specific objects. This can result in unsupported criteria when using local attributes. For example, in a database with attributes of different derivations of names beginning with 't', the following occurs:

```
MQL<43>add attr t3 type integer;
MQL<44>add attr t4 type string;
MQL<45>list attr t*;
t3
t4
MQL<46>add type typ1 attr t4;
MQL<47>add policy pol1 type typ1 state s1;
MQL<48>add bus typ1 aaa 1 policy pol1 vault v1 t4 abc;
MQL<49>temp query bus typ1 * * where 'attribute[t*] ~~ "*b*"';
Warning: #1500802: Different attribute types are found in expanded
attribute list
```

Multi-Value and Range-Value Attributes

Attributes can have multiple values and can also specify a range of values.

Multi-value attributes also support the existing concept of ranges.

Multi-Value Attributes

Traditionally, an attribute can specify only one value. However, in some cases it may be necessary for an attribute to have multiple values, which could be stored as comma-separated values. This would require additional logic in the form of a workaround in order to parse the comma-separated values and display them properly. However, this could in turn lead to issues when searching or indexing the data, as the kernel might not be aware of the workaround convention.

To deal with these issues, MQL provides the following support for multi-value attributes:

- Multi-value attributes can be applied to any attribute type, including date, integer, real, boolean, and string.
- Multi-value attributes can define an order. An order number is used to display the attribute values in a specific order.
- Dimension can be applied to the numeric fields.
- Each value can have its own unit of measure when Unit Of Measure (UOM) is applicable.
- Searching is performed against each value separately to determine a match.
- A multi-value attribute can have more than one instance of the same value.

You define an attribute to be multi-value by specifying its type as `multivalue` at the time of creation. You can also provide an order number when setting a multi-value attribute so that its value is positioned correctly.

To define an attribute as multi-value, use the command:

```
add attribute NAME multivalue;
```

To define an attribute as single-value, use one of the commands:

```
add attribute NAME notmultivalue;
add attribute NAME !multivalue;
add attribute NAME;
```

By default, attributes are single-value. It is not allowed to specify default values for multi-value attributes. Import and export can handle multi-value attribute definitions and instances.

The lxString table indexes have been changed as follows:

- The index on lxType, lxVal, lxOid is now only LxType, lxVal. In addition, the index order is lxType, lxVal on all databases.
- The unique index on lxOid, lxType includes the non-key column lxVal on SQL Server.

For more information, see *MQL Command Reference: attribute Command* in the online user assistance.

Range-Value Attributes

While multi-value attributes allow you to enter distinct values, range-value attributes allow you to enter maximum and minimum values (for example, to model a range of 100%-20%). This capability is enabled for numeric fields as well as dates (for example, to model a start and end date).

MQL provides the following support for range-value attributes:

- Range-value attributes can be applied to numeric and date attributes.
- Dimension can also be applied to the numeric fields.
- The range can have only one UOM value.

- Searching is performed against both values.
- When setting values for a range-value attribute, you can specify either endpoint as inclusive (by default), exclusive (using the keywords minexclude and maxeclude), or open-ended (if no value is specified).

You define an attribute as having a range by specifying its type as `rangevalue` at the time of creation. Only numeric and date attribute types can be defined as range-value attributes. If you attempt to define a string or boolean attribute type as `rangevalue`, an error will be thrown.

To define an attribute as specifying a range of values, use the command:

```
add attribute NAME rangevalue;
```

To define an attribute as not specifying a range, use one of the commands:

```
add attribute NAME notrangevalue;
add attribute NAME !rangevalue;
add attribute NAME;
```

By default, attributes do not specify a range of values. It is not allowed to specify default values for range-value attributes. Import and export can handle range-value attribute definitions and instances.

For more information, see *MQL Command Reference: attribute Command* in the online user assistance.

Modifying Multi-Value Attributes

To modify an existing attribute to become multi-value, use the command:

```
modify attribute NAME multivalue;
```

Only existing `singlevalue` attributes can be modified to become `multivalue`. If the attribute is already a `rangevalue`, an error will be thrown.

If there are instances of a `singlevalue` attribute in the database, the modification moves all instances from the existing attribute tables to the new multi-/range-value attribute tables in the database.

If an attribute is already `multivalue` and has multiple values (instances), changing it to `singlevalue` is not allowed and will cause an error to be thrown. For example, you can use either of the following commands to change a `multivalue` attribute to `singlevalue`:

```
modify attribute NAME notmultivalue;
modify attribute NAME !multivalue;
```

The above is OK if the attribute is already either `multivalue` or `rangevalue` and instances exist.

```
modify attribute NAME rangevalue;
```

The above is OK if the attribute is not already `multivalue`. In other words, if it is a `singlevalue` attribute with no instances, then this is permitted.

```
modify attribute NAME notrangevalue;
modify attribute NAME !rangevalue;
```

The above is OK if the existing `rangevalue` attribute does not have any instances in the database. If the attribute is already specified as `multivalue`, an error will be thrown. The following table shows a summary of what is permitted and what is not:

Modification	Allowed?
`singlevalue` (without any instances) -> `rangevalue`	Yes
`singlevalue` (with instances) -> `rangevalue`	No
`singlevalue` (without any instances) -> `multivalue`	Yes
`singlevalue` (with instances) -> `multivalue`	Yes
`rangevalue` (without any instances) -> `singlevalue`	Yes
`rangevalue` (with instances) -> `singlevalue`	No
`rangevalue` (without any instances) -> `multivalue`	No
`rangevalue` (with instances) -> `multivalue`	No
`multivalue` (without any instances) -> `singlevalue`	Yes
`multivalue` (with instances) -> `singlevalue`	Yes (as long as there is only one value for each instance)
`multivalue` (without any instances) -> `rangevalue`	No
`multivalue` (with instances) -> `rangevalue`	No

Adding/Modifying Business Objects/Connections for Multi-Value and Range-Value Attributes

Once a multi-value or range-value attribute has been added to a type or relationship, you can add business objects and connections with attribute values. You can provide a comma-separated list of values for a multi-value attribute using one of the following methods:

```
add bus T N R ATTR_NAME 1,2,3;
mod bus T N R ATTR_NAME 3,4,5;
mod bus T N R ATTR_NAME test1,test2,test3;
```

If the values themselves have spaces or commas, then they must be passed in single or double quotes using one of the following methods:

```
mod bus T N R ATTR_NAME 'foo bar1','foo bar2','foo bar3';
mod bus T N R ATTR_NAME "foo bar1","foo bar2","foo bar3";
```

Attempting to provide a comma-separated list of values for an attribute that is not multi-value will cause an error to be thrown.

For range-value attributes, the keywords `minexclude` and `maxexclude` indicate that the range endpoints do not include the minimum and maximum values, respectively. This is similar to interval notation in algebra.

Range-value attributes can have the following possible values:

Values	Description
minval::maxval For example: 5::10	• Both minval and maxval are included for any query operation. In the example, it considers the value of the range from 5 to 10 (i.e., including 5 and 10). • The selectables `includeminval` and `includemaxval` both return TRUE.
minval::maxval minexclude For example: 5::10 minexclude	• Only maxval is included for any query operation. In the example, it considers the value of the range from 6 to 10 (excluding 5). • The selectable `includeminval` returns FALSE.
minval::maxval minexclude maxexclude For example: 5::10 minexclude maxexclude	• Both minval and maxval are excluded for any query operation. In the example, it considers the value of the range from 6 to 9 (excluding 5 and 10). • The selectables `includeminval` and `includemaxval` both return FALSE.
::maxval [|maxexclude] For example: ::10	• The value of the range is considered from the negative system limit for integer or float values (e.g., from -2,147,483,648 to +10). • The selectable `openminval` returns TRUE.
minval:: [|minexclude] For example: 5::	• The value of the range is considered from minval up to the positive system limit for integer or float values. • The selectable `openmaxval` returns TRUE.

If a business object has to set a range-value attribute to have open-ended minimum and maximum values, then one of the values does not need to be provided.

The following table shows some examples of adding/modifying range-value attributes for a business object.

`add bus T N R intattr 1::3 minexclude maxexclude;`	The attribute `intattr` is stored in the database as two separate table rows with values 1 and 3, and orders 1 and 2, respectively.
`add bus T N R intattr ::3;`	The minval is open in this case.
`add bus T N R intattr 3::;`	The maxval is open in this case.
`mod bus T N R intattr add 4::5;`	Multiple values are not allowed for range-value attributes. Since the attribute already has a range of 1::3, attempting to add a second range will throw an error.
`mod bus T N R intattr 4::5;`	This range will replace the existing range value of 1::3 with the range value of 4::5.

The following is an example of using a combination of single-value, multi-value, and range-value attributes together with other business objects in a single MQL command:

```
mod bus T N R singleattr 2 multiattr test1,test2,test3 rangeattr 1::5
  policy p multiattr add test4;
```

The above command has the following effect:

1. The single-value attribute named `singleattr` is assigned the value 2.
2. The multi-value attribute named `multiattr` is assigned the values `test1,test2,test3`.
3. The range-value attribute named `rangeattr` is assigned the range 1::5.
4. The policy `p` is assigned to the business object.

A fourth value of `test4` is added to the multi-value attribute named `multiattr`.

Removing Values from Multi-Value Attributes

You can remove a value from a multi-value attribute, for example, as follows:

```
mod bus T N R remove ATTR_NAME 3;
```

The above command removes the value 3 from the multi-value attribute on the business object. If there are multiple rows with the same value, then all of those rows are removed.

A multi-value attribute can have more than one instance of the same value. The command `add bus T N R intattr 1,2,3,1;` will create four separate rows in the database table.

The following table shows some examples of adding/modifying/removing multi-value attributes for a business object. The same applies to attributes for relationships. In these examples, `intattr` is an integer, multi-valued attribute. The order values in the database are 1-based.

`add bus T N R intattr 1,2,3;`	Attribute values 1,2,3 are added in that order.
`mod bus T N R intattr add 4 order 2;`	Attribute value 4 is inserted in the third position, so that the values are 1,2,4,3.
`mod bus T N R intattr add 8;`	Attribute value 8 is added in the last position, so that the values become 1,2,4,3,8.
`mod bus T N R intattr remove 4;`	The value 4 is removed from the attribute. If there are several instance of the same value, all are removed.
`mod bus T N R intattr remove order 2;`	The value in the third position is removed.
`mod bus T N R intattr replace 5 order 2;`	The current value in the third position is replaced with the value 5.
`mod bus T N R intattr 7;`	The attribute value 7 replaces all other values, and the attribute value for the object becomes only 7.

Querying a Business Object for Multi-Value or Range-Value Attributes

Printing a business object returns multiple values for a multi-value attribute, in ascending order. For example, if there exists a type T1 with an integer multi-value attribute M1 and there is one object of this type, printing this object produces the following results:

```
add bus T1 test 0 policy P1 M1 1,2,3;
print bus T N R select attribute[M1].value;
  attribute[M1].value = 1
  attribute[M1].value = 2
  attribute[M1].value = 3
```

There is no selectable that can retrieve all of the values of a multi-value attribute as a single string.

For range-value attributes, the existing `value` selectable returns two values, as shown below:

```
add bus T1 test 0 policy P1 M1 1::3;
print bus T N R select attribute[M1].value;
  attribute[M1].value = 1
  attribute[M1].value = 3
```

The `minval` and `maxval` selectables for business objects print the minimum and maximum values, respectively, of range-value attributes. For example:

`print bus T N R select attribute[NAME].minval;`	Returns the minimum value of a range-value attribute. Returns nothing for multi-value and single-value attributes.
`print bus T N R select attribute[NAME].maxval;`	Returns the maximum value of a range-value attribute. Returns nothing for multi-value and single-value attributes.
`print bus T N R select attribute[NAME].size;`	Returns 2, since there are two instances of a range-value attribute on the business object.

Using the `minval` and `maxval` selectables in query Where clauses may have performance issues. It is recommended not to use either of these selectables as part of Where clauses.

Given a business object with a multi-value attribute M1 created with the command:

```
add bus T1 test 0 policy P1 M1 "1","2","3";
```

The following table lists the outputs produced by various queries of this business object:

Query	Returns:
`temp quer bus T1 * * where attribute[M1].value >= 1;`	`T1 test 0`
`temp quer bus T1 * * where attribute[M1].value == 1;`	`T1 test 0`
`temp quer bus T1 * * where "attribute[M1].value == 1 || attribute[M1].value == 2";`	`T1 test 0`
`temp quer bus T1 * * where attribute[M1].value > 3;`	Nothing

Given a business object with a range-value attribute M1 created with the command:

```
add bus T1 test 0 policy P1 M1 1:3;
```

The results from greater-than '>' and less-than '<' operators for range-value attributes are interpreted such that the entire range for the attribute should be '>' or '<' the literal.

The following table lists the outputs produced by various queries of this business object:

Query	Returns:
`temp quer bus T1 * * where attribute[M1].value >= 1;`	`T1 test 0` Both the minval and the maxval are >= 1.
`temp quer bus T1 * * where attribute[M1].value == 1;`	`T1 test 0` Minval is == 1.
`temp quer bus T1 * * where "attribute[M1].value == 1 || attribute[M1].value == 2";`	`T1 test 0`
`temp quer but T1 * * where attribute[M1].value > 0;`	`T1 test 0` Both minval and maxval are > 0.
`temp quer bus T1 * * where attribute[M1].value > 3;`	Nothing Both minval and maxval are <= 3.
`temp quer bus T1 * * where attribute[M1].value < 2;`	Nothing Maxval > 2.

Given a business object with a multi-value attribute M1 created with the commands:

```
add bus T1 test 0 policy P1 M1 1,2,3;
add bus T1 test2 0 policy P1 M1 4,5,6;
add bus T1 test3 0 policy P1 M1 7,2,3;
```

The following table lists the outputs produced by various queries of this business object:

Query	Returns:

`temp quer bus T1 * * where attribute[M1].value >= 1;`	`T1 test 0`
	`T1 test2 0`
	`T1 test3 0`

`temp quer bus T1 * * where attribute[M1].value == 1;`	`T1 test 0`

`temp quer bus T1 * * where attribute[M1].value != 1;`	`T1 test 0`
	`T1 test2 0`
	`T1 test3 0`

`temp quer bus T1 * * where attribute[M1].value > 3;`	`T1 test2 0` `T1 test3 0`

Given a business object with a multi-value attribute M1 created with the commands:

```
add bus T1 test 0 policy P1 M1str val1,val2,val3;
add bus T1 test2 0 policy P1 M1 val4,val5,val6;
add bus T1 test3 0 policy P1 M1str Val1,val2,val3;
```

The following table lists the outputs produced by various queries of this business object:

Query	Returns:
`temp quer bus T1 * * where attribute[M1].value == val1;`	`T1 test 0`

`temp quer bus T1 * * where attribute[M1].value != val1;`	`T1 test 0`
	`T1 test2 0`
	`T1 test3 0`
	All three objects have values other than "val1".

`temp query bus T1 * * where attribute[M1].value match val1;`	`T1 test 0` This is a case-sensitive match.

`temp query bus T1 * * where attribute[M1].value nmatch val1;`	`T1 test 0`
	`T1 test2 0`
	`T1 test3 0`
	This is a case-sensitive not-match.

`temp query bus T1 * * where attribute[M1].value smatch val1;`	`T1 test 0`
	`T1 test3 0`
	This is a case-insensitive match. "Val1" is equal to "val1" when the comparison is case-insensitive.

`temp query bus T1 * * where attribute[M1].value nsmatch 1;`	`T1 test 0`
	`T1 test2 0`
	`T1 test3 0`
	All three objects have values other than "val1".

In the case of multi-value attributes, using ".value" in the Where clause (e.g., `where attribute[M1].value == 'val1'`) is less efficient than without (e.g., `where attribute[M1] == 'val1'`). This is because the Where clause condition is resolved by means of a precise select statement, and when ".value" is not used, evaluation is done in memory, which is more intensive. It is, therefore, advisable to construct Where clauses without ".value".

Expressions with Multi-Value Selectables

Starting in V6R2013x, expressions with multi-value selectables in the Where clause maintain the data type of the underlying data. Prior to this release, multi-valued selectables in expressions were compared as strings. For example, given the following scenario:

```
businessobject SynthPart Alternate Alternating Assembly 0
attribute[SynthReal1] = -304.66
to[SynthExpand].from.attribute[SynthReal1] = -317.72
to[SynthExpand].from.attribute[SynthReal1] = 631.53
businessobject SynthPart Electronic Shielding Terminal 0
attribute[SynthReal1] = 341.34
to[SynthExpand].from.attribute[SynthReal1] = 125.79
to[SynthExpand].from.attribute[SynthReal1] = 711.95
```

The function that retrieves the data, `ConstPatternOperand::getData(int\real)`, had been incorrectly implemented. It simply called `atoi/atof` on the complete pattern string. In the above example, the pattern `-317.72\n631.53` would have been converted to a real as -317.72, and 631.53 would not even have been considered. Since -304.66 is greater than -317.72, the following query would not have returned this business object prior to V6R2013x:

```
MQL<n>temp query bus SynthPart * * orderby name where
'(attribute[SynthReal1] < to[SynthExpand].from.attribute[SynthReal1])'
select attribute[SynthReal1]
to[SynthExpand].from.attribute[SynthReal1];
```

Similarly, `125.79\n711.95` would have been converted to 125.79, which is less than 341.34, and would not have returned this business object. In V6R2013x and later, data-typed comparisons are correctly performed.

Also, multi-valued selectables in expressions were previously treated like an OR (i.e., the expression evaluated to true if any of the multiple values made it true).

In cases where there are multiple expressions involving the same attribute (e.g., where `[(attribute[M1] == val1) && (attribute[M1] == valN) .. ]`, only the first expression `(attribute[M1] == val1)` is SQL'ized, and the rest are resolved in memory.

In Where Used queries, it is prohibited to use multi-value or range-value attributes. The Orderby clause is also not allowed if multi-value or range-value attributes are used in a query.

In the case of range-value attributes, some operators (mainly the ones dealing with strings and patterns such as match, nmatch, smatch, etc.) do not produce accurate results. They treat the value `[|(minval::maxval)|]` as a string and make a comparison against it.

Dimensions and Multi-Value/Range-Value Attributes

Dimensions can be added for multi-value attributes, as is currently possible in the system (see *Dimensions*). Only one dimension is allowed per attribute type, whether multi- or single-value. However, for multi-value attributes, every single value can have its own UOM when a dimension is applied to the attribute type (i.e., each attribute value can have its own UOM value). There is a one-to-one correspondence between the entries in the attribute table and in the unit table. For example:

```
add dimension d;
mod dimension d add unit km label kilometers unitdescription
  kilometer_desc multiplier 1000 offset 0 setting METRIC true;
add attribute M1 type integer dimension d multival true;
add attribute R1 type integer dimension d rangeval true;
add type T1 attribute M1,R1;
add bus T1 N1 0 M1 "100km","200km","300km" R1 "10 km::100 km";
```

Since the attribute M1 has three values, the above `add bus` statement should see three separate entries for the UOM values in the unit table and three corresponding values in the attribute table in the database. The entries in the two tables are tied together with the order number.

The following attribute selectables return multiple values, all of which are output based on the order number:

```
attribute.inputvalue;
attribute.inputunit;
attribute.dbvalue;
attribute.dbunit;
attribute.unitvalue;
```

```
attribute.generic;
```

For range-value attributes, there is just one UOM value for the entire range in the unit table.

History and Multi-Value/Range-Value Attributes

The history entry for a multi-value attribute is delimited by a comma and truncated to 255 characters maximum length. If a single attribute setting sets the attribute to multiple values, which concatenates the values delimited by ',' and generates a string that is longer than 255 characters, only the first 255 characters of the attribute value are stored in the history tables in the database.

The history entry for the attribute modifications also includes the existing value of the attribute, and the same 255 character limitation applies to the existing attribute values.

For a multi-value attribute, an example of a history entry would be:

```
history = modify - user: creator time: Fri Jul 20, 2012 11:00:54 AM
  EDT state: s1 attr: 1,2,3,[..up to 255 chars] was 20,30,40[..up to
  255 chars]
```

For a range-value attribute, the range is specified. An example of a history entry would be:

```
history = modify - user: creator time: Fri Jul 20, 2012 11:00:54 AM
  EDT state: s1 attr: 20::30 was 10::20
```

Triggers and Multi-Value/Range-Value Attributes

The new ATTRTYPEKIND macro specifies whether an attribute is single-value, multi-value, or range-value, respectively, by having one of three values:

- Single
- Multi
- Range

The existing NEWATTRVALUE macro can be used to enter *new* multi-value and range-value attribute values delimited by "^G" (the beep character). The trigger would then have to parse the string to get the multiple values for such attributes.

The existing ATTRVALUE macro can be used to enter *current* multi-value and range-value attribute values, again delimited by "^G" (the beep character). The trigger would then have to parse the string to get the multiple values for such attributes.

Unique Keys/Indexes and Multi-Value/Range-Value Attributes

Unique keys and indexes are not supported for multi-value or range-value attributes. This means that:

- If a multi-value or range-value attribute is used as a field while defining an index or unique key, an error will be thrown.
- If there is an existing unique key or index that has a single-value attribute and you try to modify that attribute to become a multi-value or range-value attribute, an error will be thrown.

Adaplets and Multi-Value/Range-Value Attributes

Adaplets do not support multi-value or range-value attributes. This means that:

- If you modify an existing attribute to become a multi-value or range-value attribute, the first time that the attribute vault is loaded after the change, an error will be thrown.
- If any new attributes are added that are multi-value or range-value attributes and they are used in an adaplet mapping file, again an error will be thrown.

Dimensions

The dimension administrative object provides the ability to associate units of measure with an attribute, and then convert displayed values among any of the units defined for that dimension. For example, a dimension of Length could have units of centimeter, millimeter, meter, inch and foot defined. Dimensions are used only with attributes. For more information, see *MQL Command Reference: attribute Command: Add Attribute Command: Dimension Clause* in the online user assistance.

The definition of the units for a dimension includes determining which unit will be the default (the normalized unit for the dimension), and the conversion formulas from that default to the other units. The conversion formulas are based on a multiplier and offset entered when the unit is defined. The normalized unit has a multiplier of 1 and an offset of 0.

To convert to the normalized value stored in the database to a different unit, the system uses this formula:

```
normalized value = unit value * multiplier + offset
```

To display a value in units other than the normalized units, the system uses this formula:

```
unit value = (normalized value - offset) / multiplier
```

Only the normalized value is stored in the database. When an application requires the value for an attribute to be displayed, the system converts the normalized value to the units required. The value can be entered in any supported unit of the dimension, but it will be converted and stored in the default units.

Real attribute normalized values are stored with the same precision as real attribute values with no dimension applied. See *Attributes* for more information. To avoid round-off errors with integer attributes, the default units should be the smallest unit (for example, millimeters rather than centimeters or meters).

The conversion process affects the precision. In general, up to 12 digits of precision can be assumed. For each order of magnitude that the offset and the converted value differ, another digit of precision is lost.

Dimensions help qualify attributes that quantify an object. For example, for a type with an attribute Weight, the user needs to know if the value should be in pounds or kilograms, or another dimension of weight. When the attribute definition includes a dimension, the user is provided with that information in the user interface. In addition, the user has the ability to choose the units of the dimension to enter values.

When applying dimensions to attributes that already belong to business object instantiations, refer to *Applying a Dimension to an Existing Attribute* for information on converting the existing values to the required normalized value.

Defining a Dimension

Before attributes can be defined with a dimension, the dimension must be created. You can define a dimension if you are a business administrator with Attribute access, using the add dimension command:

```
add dimension NAME [ADD_ITEM {ADD_ITEM} ];
```

`NAME` is the name you assign to the dimension.

`ADD_ITEM` is an Add Dimension clause that provides additional information about the dimension you are creating.

For more information, see *MQL Command Reference: dimension Command* in the online user assistance.

Choosing the Default Units

Before you define a dimension, you need to decide which units of that dimension will be the default. That unit will have a multiplier of 1 and an offset of 0. You must calculate the multiplier and offset values for all other units of the dimension based on the default.

For example, the following table shows the definition for a Temperature dimension normalized on Fahrenheit:

Unit	Label	Multiplier	Offset
Fahrenheit	degrees Fahrenheit	1	0
Celsius	degrees Celsius	1.8	32

If you wanted to normalize the dimension on Celsius, you would enter these values when defining the units:

Unit	Label	Multiplier	Offset
Fahrenheit	degrees Fahrenheit	.555555555555555	17.7777777777777777
Celsius	degrees Celsius	1	0

For dimension definitions that are to be applied to an integer, all multiplier and offset values in the dimension should be whole numbers, and the "smallest" unit should be the default.

Interfaces

You may want to organize data under more than one classification type. For instance, a Part may have a classification based on its function, which is most typical, but it may also require classification on other issues such as production process, manufacturing location, etc. For each classification type, there is typically a collection of attributes that can be defined for each instance of the classification type and used for searching.

An *Interface* is a group of attributes that can be added to business objects as well as connections to provide additional classification capabilities. When an Interface is created, it is linked to (previously-defined) attributes that logically go together. Interfaces are defined by the Business Administrator and are added to business object or relationship instances.

You must be a Business Administrator to add or modify interfaces.

An Interface can be *derived* from other Interfaces, similar to how Types can be derived. Derived Interfaces include the attributes of their parents, as well as any other attributes associated with it directly. The types or relationships associated with the parents are also associated with the child.

The primary reason to add an interface to a business object or a connection is to add the attributes to the instance that were not defined on the type. Moreover, when you add an interface to a business object or a connection, it gives you the ability to classify it by virtue of the interface hierarchy.

Attribute values that come from interfaces cannot be used in a create access rule, because the interfaces are not applied until AFTER the create access checks.

Defining an interface

An object interface is created with the Add Interface command:

```
add interface NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the interface.

`ADD_ITEM` is an Add Interface clause which provides additional information about the interface you are creating.

For more information, see *MQL Command Reference: interface Command* in the online user assistance.

Types

A *type* identifies a kind of business object and the collection of attributes that characterize it. When a type is created, it is linked to the (previously defined) attributes that characterize it. It may also have methods and/or triggers associated (refer to *Programs* and the *Configuration Guide: Triggers*). Types are defined by the Business Administrator and are used by users to create business object instances.

A type can be *derived* from another type. This signifies that the derived type is of the same kind as its parent. For example, a Book is a kind of Publication which in turn is a kind of Document. In this case, there may be several other types of Publications such as Newspaper, Periodical, and Magazine.

This arrangement of derived types is called a *type hierarchy.* Derived types share characteristics with their parent and siblings. This is called *attribute inheritance*. When creating a derived type, other attributes, methods, and triggers can be associated with it, in addition to the inherited ones. For example, all Periodicals may have the attribute of Page Count. This attribute is shared by all Publications and perhaps by all Documents. In addition, Periodicals, Newspapers, and Magazines might have the attribute Publication Frequency.

You must be a Business Administrator to add or modify types. (Refer also to your Business Modeler Guide.)

Type Characteristics

Implicit and Explicit

Types use explicit and implicit characteristics:

- *Explicit characteristics* are attributes that you define and are known to 3DSpace.
- *Implicit characteristics* are implied by the name only and are known only to the individual user.

For example, you may create a type called “Tax Form” which contains administrator-defined explicit attributes such as form number, form type, and tax year. Or, Tax Form may contain no explicit attributes at all.

When a type exists without administrator-defined attributes, it still has implicit characteristics associated with it. You would know a tax form when you saw it and would not confuse it with a type named “Health Form.” But the characteristics you use to make the judgment are implicit—known only by you and not 3DSpace.

Inherited Properties

Types can inherit properties from other types:

- *Abstract types* act as categories for other types.

  Abstract types are not used to create any actual instances of the type. They are useful only in defining characteristics that are inherited by other object types. For example, you could create an abstract type called Income Tax Form. Two other abstract types, State Tax Form and Federal Tax Form, inherit from Income Tax Form.

- *Non-abstract types* are used to create instances of business objects.

  With non-abstract types, you can create instances of the type. For example, assume that Federal Individual Tax Form is a non-abstract type. You can create business objects that contain the actual income tax forms for various individuals. One object might be for a person named Joe Smith and another one for Mary Jones. Both objects have the same type and characteristics although the contents are different based on the individuals.

Defining a Type

An object type is created with the Add Type command:

```
add type NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the type.

`ADD_ITEM` is an Add Type clause which provides additional information about the type you are creating.

For more information, see *MQL Command Reference: type Command* in the online user assistance.

Formats

A *format* definition is used to capture information about different application file formats. A format stores the name of the application, the product version, and the suffix (extension) used to identify files. It may also contain the commands necessary to automatically launch the application and load the relevant files from Collaboration and Approvals. Formats are the definitions used to link Collaboration and Approvals to the other applications in the users' environment.

Applications typically change their internal file format occasionally. Eventually older file formats are no longer readable by the current version of the software. It is wise to create new format definitions (with appropriate names) as the applications change so that you can later find the files that are in the old format and bring them up to date.

The system does not do any checking of the type of file that is checked into a format. For example, a word document with a .doc extension can be checked into a format defined for HTML files. This means that formats can be used to define directories for the checked in files of a business object.

A business object can have many file formats and they are linked to the appropriate type definition by the policy definition (see *Policies*).

You must be a Business Administrator to add or modify formats.

Defining a Format

Format definitions are created using the MQL Add Format command. This command has the syntax:

```
add format NAME [ADD_ITEM {ADD_ITEM}]
```

`NAME` is the name you assign to the format.

`ADD_ITEM` is an Add Format clause which provides more information about the format you are creating.

For more information, see *MQL Command Reference: format Command* in the online user assistance.

Policies

A *policy* controls a business object. It specifies the rules that govern access, approvals, lifecycle, revisioning, and more. If there is any question as to what you can do with a business object, it is most likely answered by looking at the object's policy.

You must be a Business Administrator to add or modify policies. (Refer also to your Business Modeler Guide.)

A policy is composed of two major sections: the first describes the general behavior of the governed objects and the second describes the lifecycle of the objects.

General Behavior

The first section controls the creation of the object and provides general information about the policy. This information includes:

- The types of objects the policy will govern.
- The types of formats that are allowed.
- The default format automatically assigned.
- Where and how checked in files are managed.
- How revisions will be labeled.

Lifecycle

The second section provides information about the lifecycle of the objects governed by the policy. A lifecycle consists of a series of connected states, each of which represents a stage in the life of the governed objects. Depending on the type of object involved, the lifecycle might contain only one state or many states. The purpose of the lifecycle is to define:

- The current state of the object.
- Who will have access to the object.
- The type of access allowed.
- Whether or not the object can be revised.y
- Whether or not files within the object can be revised.
- The conditions required for changing state.
- Whether or not the state has been published. See "Published States" on page 71.

A policy can exist without any states defined. This is enabled mainly to support legacy data. There are certain disadvantages to not defining any states:

- It is not possible to have access control without any state definition.
- Some operations, like disabling checkout history, cannot be performed since no state exists to control this.

If a policy is created without state, a warning is displayed stating "Policy has no STATE defined."

Determining Policy States

When creating a policy, defining the policy states is most often the most difficult part. How many states does the policy need? Who should have access to the object at each state and what access should each person have at each state? Which access takes precedence over the other? Should you allow revisions at this state? Should you allow files to be edited? What signatures are required to move the object from one state to another? Can someone override another's signature? As described below, all of these questions should be answered in order to write the state definition section of a policy.

How Many States are Required?

A policy can have only one state or many. For example, you might have a policy that governs photographic images. These images may be of several types and formats, but they do not change their state. In general, they do not undergo dramatic changes or have stages where some people should access them and some should not. In this situation, you might have only one state where access is defined.

Let's examine a situation where you might have several states. Assume you have a policy to govern objects during construction of a house. These objects could have several states such as:

State	Description
Initial Preparation	The building site is evaluated and prepared by the site excavator and builder. After the site is reviewed and all preparations are completed, the excavator and builder sign off on it and the site enters the second state, Framing.
Framing	Carpenters complete the framing of the house and it is evaluated by the builder, architect, and customer. In this state, you may want to prohibit object editing so that only viewing is allowed. If the framing is complete to the satisfaction of the builder, architect, and customer, it is promoted to the third state, Wiring.
Wiring	The electrician wires the house. However, the electrician may sign off on the job as completed only to have the builder reject it. When approval is rejected, promotion to the next state is prevented from taking place.

As the house progresses through the building states, different persons would be involved in deciding whether or not the object is ready for the next state.

When determining how many states an object should have, you must know:

- What are the states in an object's life.
- Who requires access to the object.
- What type of access they need.

Once a policy is defined, you can alter it even after business object instances are created that are governed by it.

Who Will Have Object Access?

There are three general categories used to define who will have access to the object in each state:

- Public—refers to everyone in the database. When the public has access in a state, any defined user can work with the business object when it is in that state.

- Owner—refers to the specific person who is the current owner of the object instance. When an object is initially created in the database, the person who created it is identified as the *owner* of the object. This person remains the owner unless ownership is transferred to someone else.
- User—refers to a specific person, group, role, or association who will have access to the object. You can include or exclude selected groupings or individuals when defining who will have access.

For additional information on access privileges, including which access takes precedence over the other, see *Studio Modeling Platform Overview: Controlling Access* in the online user assistance.

Rules to Determine Access

You can specify the rules for a policy or state to define specific accesses. A rule can contain one or many rule items that define accesses for a username, or a username+key. The policy or state can reference all rule items or specific rule items.

If you reference all the rule items (implies no mapping), all the rule items in that rule are added to the policy or state as-is, using the same usernames and the same keys. The owner and public rules with no key are not referenced.

If you reference specific rule items (allows full mapping), the policy or state definition identifies which rule items to use based on the username and key. If you do not specify a key, then only rule items that match that username and have no defined key are referenced. Owner and public rule items that have a key can be referenced, but the built-in owner and public rules (no key) cannot be referenced.

An error occurs if any rule items use a username/key already defined for the policy or state, including any owner and public rules.

For more information about the syntax for defining policies and states, see *MQL Command Reference: policy Command: Add Policy: State Command* in the online user assistance. For more information about rules, see *Rules*.

Is the Object Revisionable?

In each state definition are the terms *Versionable* and *Revisionable*. The term Revisionable indicates whether a new Revision of the object can be made. Versionable is not used at this time, and setting it has no affect on policy behavior.

You can decide when in the object's lifecycle revisions are allowed by setting the switch ON or OFF in each state definition. This setting is independent of who (which person, role or group) has access to perform the operations.

Published States

Every state has a published flag. In policies where this flag is turned on, a best-so-far (BSF) flag is automatically propagated to business objects that enter such states. By default, the published flag is turned off. A migration tool is available to migrate policies, both to incorporate the new published flag and to update all business objects governed by policies that have the flag turned on in order to set the published flag correctly. See the *Database Migration Guide*, available in the Program Directory, for details on the migration tool.

The definition of a policy accepts that each of its states can be marked as *published*. You can set this flag on the state administrative object by invoking the following MQL command:

```
mod policy NAME state STATE published TRUE|FALSE;
```

To determine whether the published flag has been set on an object, you can use:

```
print policy NAME select state.published;
```

To retrieve the boolean value of the published flag as it is propagated to business objects, use:

```
print bus T N R select.current.published;
```

Policies allow definition of two revision sequences, major and minor. You can define a revision sequence for a policy as follows:

```
add policy NAME majorsequence A,B,C,... minorsequence 1,2,3
  delimiter '-';
```

If both major and minor sequences are defined, then a delimiter is required. The delimiter is used to concatenate major and minor revision strings for storage in the database, so it must be an ASCII non-alphanumeric character that could never be part of a major/minor revision string calculated from the two sequences.

The Modify Policy command allows you to edit either sequence, but not to change the delimiter or add/remove either sequence, as this would make major/minor revision strings impossible to parse. You can use the Transition Revisions command to add a second sequence to a policy that only one (see the *Database Migration Guide* for details).

The Print Policy command also allows the corresponding selectables, such as:

```
print policy NAME select majorsequence minorsequence delimiter;
```

Policy and state access definitions have the following access flags:

- *minorrevise* is synonymous with the earlier *revise* flag, and is used to control the Revise Minor command for adding a new minor revision.
- *majorrevise* is used to control the Revise Major command for adding a new major revision.

The following events support type definitions:

- *minorrevision* is synonymous with the earlier *revision* keyword, used for defining triggers on a type.
- *majorrevision* has been added as a new event for defining triggers on a type.

Minorrevision and majorrevision are values that are derived from the revision field using the policy delimiter. Object uniqueness that incorporates revision information must include both minor and major revision fields. The revision keyword should be used to define such a uniquekey. Therefore, minorrevision and majorrevision are not supported as fields for unique keys.

These capabilities in policy definition have corresponding pieces of data and selectables for business objects.

You can mark any state in a policy to indicate whether it is allowed to create a major/minor revision of an object in that state as follows:

```
mod policy NAME state STATE minorrevision TRUE|FALSE
  majorrevision TRUE|FALSE;
```

The terms "minorrevise" and "revise" are synonymous when defining access rules in policy and in the `print bus select current.access[minorrevise]` command. However, as output from the Print Policy command (e.g., `print policy select state` and `print bus select current.access`), the earlier "revise" keyword continues to be used for forward compatibility. Likewise, the keyword "revisioned" continues to be used to mark minorrevise events in history.

Input Keywords

In the following cases, "revise" and "revision" have the same meaning as long as the object's policy supports ONLY minor revisioning. If, however, the governing policy supports both major and minor revisions, there is a difference:

- `print bus T N R select revision` prints the full revision string of the object. If the policy supports both minor and major revisions, this will be major-minor.
- `print bus T N R select revisions` prints the full revision string for all objects in the object's minor revision sequence.
- `copy/mod bus T N R name NEWNAME revision NEWREV: NEWREV` uses the full revision string (major-minor) as specified in the governing policy.

In an access rule filter:

- A filter-including expression of the form `current.access[ACCESS_TYPE] == TRUE` accepts either minorrevise or revise as the ACCESS_TYPE.
- A filter-including expression of the form `current.access ~~ *ACCESS_TYPE*` is risky since revise is now be a substring of both minorrevise and majorrevise (but see Output Keywords below).

Output Keywords

In various places, the kernel outputs the keywords revise/revision or populates macros with such a string. Changing such output could break any application code that depends on the current 'revise/revision' keyword. Although it would be cleaner and more consistent to migrate these outputs to minorrevise/minorrevision, the current output has been retained to satisfy forward compatibility.

- `print bus T N R select current.access` maintains 'revise' as the keyword describing minor revise access (e.g., `current.access == read,modify,revise,show`).
- The $ACCESS macro can be used in an access rule filter and is populated with the current access being checked whenever the filter is evaluated (e.g., the following rule allows a different condition for modify vs. revise access):

  ```
  ($ACCESS == modify && <modify-condition>) || ($ACCESS == revise &&
  <revise-condition>)
  ```

This continues to be populated as 'revise'.

- The EVENT macro continues to be populated as 'Revision' to pass to revision triggers.
- History records still say `history = revisioned - user: creator ….`

How Do You Change From One State to the Next?

Most often a change in state is controlled by one or more persons, perhaps in a particular role or group. For example, during the construction of a house, the customer and the builder might control the change in state. If you break the building stage down into smaller states, you might have the object's transition controlled by the site excavator, foundation expert, electrician, or plumber. As the house progresses through the building states, different persons would be involved in deciding whether the object is ready for the next state. You certainly would not want the carpenters to begin working before the foundation is done.

Signatures are a way to control the change of an object's state. Signatures can be associated with a role, group, person, or association. Most often, they are role-related. When a signature is required, a person must approve the object in order for the object to move on to the next state. If that person does not approve it, the object remains in the current state until the person does approve or until someone with higher authority provides approval.

More than one signature can be associated with the transition of an object. Lifecycles can be set up such that the signature that is approved determines which state is the next in the object's life.

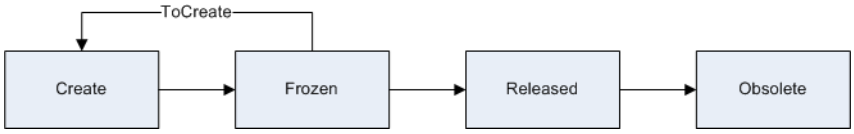
A signature can be approved or rejected. For example, an electrician could say a job is done only to have the builder reject it. When approval is rejected, promotion to the next state is prevented from taking place.

Filters can be defined on a signature requirement to determine if it is fulfilled. If the filter evaluates to true, then the signature requirement is fulfilled. This is useful for adding required signatures that are conditional, dependent on some characteristic of a business object.

In the sections that follow, you will learn more about the actual procedures to define a policy and the object states as well as the procedures that manipulate and display policy definitions.

Blocking States and Signatures

After a policy has been defined, you might find that you want to remove a state or a signature. After doing so, you can add it back in later if necessary. For example, a policy has these states and signature (in this case, ToCreate is considered a signature because it exits from the normal promotion sequence):



To remove the Frozen state, you need to block the transition from Create to Frozen and from Frozen to Released, and remove the signature ToCreate. You would use these commands to accomplish this:

```
modify policy POLICYNAME state Create block Frozen;
modify policy POLICYNAME state Frozen block Released;
modify policy POLICYNAME state Create remove signature ToCreate;
```

The result would look like this:

To add a state back in that was previously removed, you would use these commands:

```
modify policy POLICYNAME state Create unblock Frozen;
modify policy POLICYNAME state Frozen unblock Released;
```

To add the signature back in, you need to define the signature requirements for that transition. For more information, see *MQL Command Reference: policy Command: Modifying Signature Requirements* in the online user assistance.

Defining an Object Policy

Policies are defined using the Add Policy command:

```
add policy NAME [ITEM {ITEM}];
```

NAME is the name you assign to the policy.

ITEM is an Add Policy clause which defines information such as the types of objects governed by the policy, the types of formats permitted by the policy, the labeling sequence for revisions, the storage location for files governed by the policy, and the states and conditions that make up an object's lifecycle.

For more information, see *MQL Command Reference: policy Command* in the online user assistance.

Relationships

A *relationship* definition is used along with the policy to implement business practices. Therefore, they are relatively complex definitions, usually requiring planning.

For example, in manufacturing, a component may be contained in several different assemblies or subassemblies in varying quantities. If the component is later redesigned, the older design may then become obsolete. Component objects could be connected to the various assembly and subassembly objects that contain it. Each time objects are connected with this relationship, the user could be prompted for the quantity value for the relationship instance. If the component is later redesigned, the older design may become obsolete. When a revision of the component object is then created, the relationship would disconnect from the original and connect to the newer revision. If the component is cloned because a similar component is available, the cloned component may or may not be part of the assembly the original component connects to. The connection to the original should remain but there should be no connection to the cloned component.

For the process to work in this fashion, the relationship definition would include the attribute "quantity." The cardinality would be "many to many" since components could be connected to several assemblies and assemblies can contain many components. The revision rule would be "float" so new revisions would use the connections of the original. The clone rule would be "none" so the original connection remains but no connection is created for the clone.

A relationship can be *derived* from another relationship. This signifies that the derived relationship is of the same kind as its parent. This arrangement of derived relationships is called a *relationship hierarchy.* Derived relationships share characteristics with their parent and siblings. This is called *relationship inheritance*. When creating a derived relationship, other attributes, methods, and triggers can be associated with it, in addition to the inherited ones.

You must be a Business Administrator to define relationships. (Refer also to your Business Modeler Guide.) Relationships are typically initially created through MQL when all other primary administrative objects are defined. However, if a new relationship must be added, it can be created with Business Administrator.

Collecting Data for Defining Relationships

In an MQL schema definition script, relationship definitions should be placed after attributes and types. Before writing the MQL command for adding a relationship definition, the Business Administrator must determine:

- Of the types of business objects that have been defined, which types will be allowed to connect directly to which other types?
- What is the nature and, therefore, the name of each relationship?
- Relationships have two ends. The *from* end points to the *to* end. Which way should the arrow (in the Indented and Star Browsers) point for each relationship?
- What is the meaning of the relationship from the point of view of the business object on the *from* side?
- What is the meaning of the relationship from the point of view of the business object on the *to* side?
- What is the cardinality for the relationship at the *from* end? Should a business object be allowed to be on the *from* end of only one or many of this type of relationship?

- What is the cardinality for the relationship at the *to* end? Should a business object be allowed to be on the *to* end of only one or many of this type of relationship?
- When a business object at the *from* end of the relationship is revised or cloned, a new business object (similar to the original) is created. What should happen to this relationship when this occurs? The choices for revisions and clones are: `none`, `replicate`, and `float`.

  Should the relationship stay on the original and not automatically be connected to the new revision or clone? If so, pick `none`.

  Should the relationship stay on the original and automatically connect to the new revision or clone? If so, pick `replicate`.

  Should the relationship disconnect from the original and automatically connect to the new revision or clone? If so, pick `float`.
- When a business object at the *to* end of the relationship is revised or cloned, a new business object (similar to the original) is created. What should happen to this relationship when this occurs? The choices are the same as for the *from* end.
- What attributes, if any, belong on the relationship? Quantity, Units, and Effectivity are examples of attributes which logically belong on a relationship between an assembly and a component rather than on the assembly or component business object. Each instance, or use of the relationship, will have its own values for these attributes which apply to the relationship between the unique business objects it connects.

Use a table like the one below to collect the information needed for relationship definitions.

Relationship Name	_____	_____	_____	_____	_____	_____
From Type	_____	_____	_____	_____	_____	_____
From Meaning	_____	_____	_____	_____	_____	_____
From Cardinality	_____	_____	_____	_____	_____	_____
From Rev Behavior	_____	_____	_____	_____	_____	_____
From Clone Behavior	_____	_____	_____	_____	_____	_____
To Type	_____	_____	_____	_____	_____	_____
To Meaning	_____	_____	_____	_____	_____	_____
To Cardinality	_____	_____	_____	_____	_____	_____
To Rev Behavior	_____	_____	_____	_____	_____	_____
To Clone Behavior	_____	_____	_____	_____	_____	_____
Attributes	_____	_____	_____	_____	_____	_____
Dynamic Relationship?	_____	_____	_____	_____	_____	_____

Dynamic Relationships

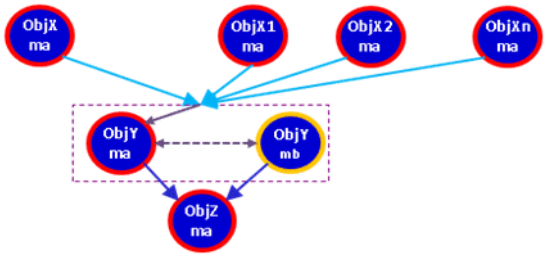
To support major/minor revisioning of business objects, "dynamic" relationships have a pointer to a MajorId representing a minor revision family rather than an individual business object. The TO end of a dynamic relationship resolves to a specific business object by identifying the best-so-far (BSF) object within the minor revision family.

When used with the Add Relationship command, the `dynamic` keyword implements support for minor-revision insensitivity. This keyword is mutually exclusive with any of the Replicate/Float/None options on the TO end, since it defines a built-in dynamic behavior of its *to* pointer.

Together with *best-so-far*, the concept of *published* objects makes it possible unambiguously to identify a single object within a minor revision family in order to resolve dynamic relationships. An object is marked "published" when it is at an appropriately mature state in its lifecycle. An unambiguous best-so-far object is then either:

- The last object in the family that is marked "published," or
- The last object in the family if none are published.

The *to* end of a dynamic relationship resolves to a specific business object by identifying the best-so-far (BSF) object within the minor revision family).



The dynamic relationship feature adds an additional requirement for database definition with Oracle installations: the Oracle user (i.e., database) must have CREATE VIEW privileges. Run the following command to add this privilege:

```
SQL> grant connect, resource, create view to USER:
Grant succeeded.
SQL> commit;
Commit complete.
```

where USER is the V6 Oracle user.

For more information, see *MQL Command Reference: relationship Command: Add Relationship: Dynamic Clause* in the online user assistance.

MQL Export/Import

Policies are exported/imported with added fields.

Business object export/import is not supported on objects governed by policies with `majorsequence` set.

Dynamic relationships are not exported with business objects.

Defining a Relationship

A relationship between two business objects is defined with the Add Relationship command:

```
add relationship NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the relationship.

`ADD_ITEM` is an Add Relationship clause which provides more information about the relationship you are creating.

For more information, see *MQL Command Reference: relationship Command* in the online user assistance.

Rules

Use rules to limit user access to attributes, forms, programs, and relationships. Unlike policies, rules control access to these administrative objects regardless of the object type or state. For example, a policy might allow all users in the Engineering group to modify the properties of Design Specification objects when the objects are in the Planning state. But you could create a rule to prevent those users from changing a particular attribute of Design Specifications, such as the Due Date. In such a case, Engineering group users would be unable to modify the Due Date attribute, no matter what type of object the attribute is attached to. For a explanation of how rules work with other objects that control access, see *Studio Modeling Platform Overview: Access Precedence* in the online user assistance.

When you create a rule, you define access using the three general categories used for assigning access in policies: public, owner, and user (specific person, group, role, or association). For a description of these categories, see *Studio Modeling Platform Overview: Controlling Access: User Access: User Categories and Hierarchies* in the online user assistance.

You can also include keys in rules to specify multiple access definitions for the same user. For more information, see *Studio Modeling Platform Overview: Controlling Access: User Access: Security Context Access in Policies and Rules* in the online user assistance.

For complete information about creating rules, see *MQL Command Reference: rule Command* in the online user assistance.

Creating a Rule

Rules are defined using the Add Rule command:

```
add rule NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the rule.

`ADD_ITEM` is an Add Rule clause which defines information about the rule including the description and access masks.

For more information, see *MQL Command Reference: rule Command* in the online user assistance.

Ownership

Business objects and relationships can have multiple owners. In addition, several objects may need to have a common baseline access level. This section describes how ownership of business objects and relationships functions, including multiple ownerships and ownership inheritance.

Multiple Ownerships

There are many cases where a business object or relationship may require multiple ownerships, such as a part that has multiple RDO, RMO, and RSO security attributes, each of which represent an organization or project owner. It is possible to specify multiple ownerships for an object. A complete ownership definition includes the following:

- Organization—a "role" object that represents an organization
- Project—a "role" object that represents a project
- Comment—a short string used primarily for annotation
- Access—a comma-separate list of security tokens

Each of these fields can be specified when providing an ownership. The first three fields (org, project, and comment) together provide a unique identifier for the ownership entry.

You can add ownerships to or subtract them from an existing business object or relationship. Ownership information is used to determine object access rights dynamically. Commands that are used to get or set the single organization/project ownership on business objects continue to function, but generate an error if more than one ownership is present.

For more information, see *MQL Command Reference: businessobect Command: Modify Business Object: Adding or Removing Business Object Ownerships* and *MQL Command Reference: relationship Command: Modify Relationship: Adding or Removing Relationship Ownerships* in the online user assistance.

Ownership Inheritance

Rather than maintaining security on a per-object basis, it is possible to govern a list of objects where the entire collection of objects has a common baseline access level by aggregating ownerships based on another object. This allows an object to inherit the ownership list from another object as its baseline access in addition to having its own direct ownership. Thus, it is no longer necessary to manage ownership by adding or subtracting ownerships individually on each object in a workspace since ownership for an object comprises an embedded reference to a parent object.

Existing applications that store parent information as a relationship (e.g., workspace) must add an ownership definition in addition to the relationship. Applications that currently lack a parent or folder notation can create the parent ownership entry without creating a new relationship entity by specifying a parent object. The access rule engine dynamically extends an entity’s ownership list to include the (non-inherited) ownerships of the parent.

To do this, you specify an object ID and another optional comment relating to the parent object. Ownerships, constructed in this form implement ownership inheritance from the parent. The access rule engine dynamically extends an entity's ownership list to include the ownerships of the parent. These inherited ownerships include all ownerships added to the entity using the multiple-ownership feature. The ownerships do not include the parent’s primary ownership identified in the parent’s organization and project basic properties. You can use ownership inheritance on multiple levels.

Object A can inherit from object B, and object B can inherit from object C. This implies that Object A also inherits from object C. For more information, see *MQL Command Reference: businessobject Command: Modify Business Object: Adding Ownership Inheritance* in the online user assistance.

Inheritance Rules

Use inheritance rules to define ownership or access details that can be assigned to an object on the to or from end of a relationship when a connection is made based on the criteria defined in the rule. For example, if you add a document to a folder, you can define an inheritance rule to add access to that document. If the document is removed from the folder, any access granted by the inheritance rule is removed from the document.

When defining an inheritance rule, you specify a relationship and whether the ownership or access operates on the object at the from or to end of the relationship. Include clauses in the rule, based on object types and policies, that define when this inheritance rule should be applied.

To define the from and to criteria for an inheritance rule, specify the object type and policy type. You can specify all for the type or policy. For example:

```
add inheritancerule SoftwareRule operateson to appliesto
ProductRelationship ownership SoftwareOwnerRI from type Part,Product
policy all to type "Software Product" policy "Software Development" as
show,read
```

In this example:

- `SoftwareRule` is the name of the inheritance rule
- `operateson to` defines which end of the relationship the specified ownership or access will be applies (in could also be `operateson from`)
- `appliesto ProductStructure` defines the relationship that when used to connect two objects, the system checks the criteria for from and to below to determine if this inheritance rule should be applied
- `ownership` defines ownership inheritance (you can also define `access` inheritance)
- `SoftwareOwnerRI` defines the name of this rule item (an inheritance rule can contain multiple rule items)
- `from type Part,Product policy all` defines the from side of a relationship
- `to type "Software Product" policy "Software Development"`defines the to side of a relationship
- `as show,read` defines the specific accesses granted by this rule

The interpretation of the above inheritance rule is that whenever an app uses the ProductStructure relationship to connect an object of type Part or Product being governed by any policy to an object of type Software Product being governed by the Software Development policy, then the shown access (show,read) is added to the ownership of the Software Product (the to side of the relationship as defined by operateson).

When the app determines that the criteria for an inheritance rule have been met, it adds a comment to the business object (as defined by the operateson and from or to clauses) in this format:

```
enorule_<rel_pid>_<tenant_id>_<rule_id>_<rule_item_id>
```

Where:

- `enorule` is the prefix that represents auto-generated inheritance
- `rel_pid` is the physical id of the connection object
- `tenant_id` is the tenant id
- `rule_id` is the inheritance rule id
- `rule_item_id` is the id of the specific rule item within the inheritance rule

Whenever a modify, delete, or reroute connection operation occurs, the system re-evaluates all the related inheritances for the affected object. For example, a connection exists between BusObject1 and BusObject2, and an inheritance rule applies to BusObject 2 based on that connection. If you move the relationship from BusObject2 to BusObject3, the inheritance rule on BusObject2 is removed and the inheritance rule is applied to BusObject3 (if the criteria of the rule are still met).

For more information, see *MQL Command Reference: inheritancerule Command* in the online user assistance.

3 Manipulating Data

This chapter explains how to create and use business objects to manipulate data.

In this section:

Creating and Modifying Business Objects

Business objects form the body of Collaboration and Approvals. They contain much of the information an organization needs to control. Each object is derived from its previously-defined type and governed by its policy. Therefore, before users can create business objects, the Business Administrator must create definitions for the types (for more information, see *Types* in Chapter 2) and policies (for more information, see *Policies* in Chapter 2) that will be used. In addition, the users (persons, groups, and roles) must be defined before they can have access to the application (for more information, see *Studio Modeling Platform Overview: Controlling Access: User Access* in the online user assistance).

When creating a business object, the first step is to define (name) the object and assign an appropriate description and attribute values for it. File(s) can then be checked into the object and it can be manipulated by establishing relationships, moving it from state to state and perhaps deleting or archiving it when it is no longer needed. This chapter describes the basic definition of the object and its attributes. In the next chapter, relationships, connections, states, checking files in and out, and locking objects are described in more detail.

Using Physical and Logical IDs

A physical id is a global identifier that is unique to each business object or relationship and a logical id is a global identifier shared by all members of a revision sequence. Physical and logical ids are required for Product Structure Design applications. You must upgrade your database to use these ids. See the Program Directory for this release.

Specifying a Business Object Name

When you create or reference a business object, you must give its full business object name. The full business object name must contain three elements:

```
TYPE NAME REVISION
```

Each element must appear in the order shown. If any element is missing or if the values are given in the wrong order, the business object specification is invalid.

You can also optionally specify the vault in which the business object is held. When the vault is specified in this manner, only the named vault needs to be checked to locate the business object. This option can improve performance for very large databases.

```
TYPE NAME REVISION [in VAULT]|ID [in VAULT]
```

See *Defining a Business Object* later in this chapter for more information about additional elements.

The full business object specification includes `TYPE NAME REV`: the type from which the object was created, the object name—the user-supplied identifier that is associated with the definition and identifies the object to the end user(s)—and the revision.

In the sections that follow, each of the three required elements is discussed and sample values are given.

Business Object Type

The first element in a business object specification is the object's type. Every object must have a type associated with it. When you specify the object's type, remember that it must already be defined.

If the type name you give in the business object specification is not found, an error message will display. If this occurs, use the List Type command (for more information, see *MQL Command Reference: list admintype Command* in the online user assistance) to check for the presence and spelling of the type name. Names are case-sensitive and must be spelled using the same mixture of uppercase and lowercase letters.

When you are assigning a type to a business object, note that all types have attributes associated with them. These attributes appear as fields when the object is accessed in Matrix Navigator. These fields can then be filled in with specific values or viewed for important information.

For example, you might assign a type of "Metallic Component" to an object you are creating. With this type, you might have four attributes: type of metal, size, weight, and strength. If you use an Attribute clause in the `add businessobject` command or modify the attributes, you can insert values that will appear whenever the object is accessed. If attributes are not specified when a business object is added or modified, the attribute defaults (if any) are used.

Business Object Name

The second element in a business object specification is the business object name. This consists of a character string that will be used to identify the business object and to reference it later. It should have meaning to you and guide you as to the purpose or contents of the business object. While the Description clause can remind you of an object's function, it is time-consuming to have to examine each object to find the one you want. Therefore, you should assign a name that clearly identifies and distinguishes the object.

You can use your exact business terminology rather than cryptic words that have been modified to conform to the computer system limitations. Collaboration and Approvals has few restrictions on the characters used for naming business objects. For more information, see *MQL Command Reference: General Syntax: About Administrative Object Names* in the online user assistance.

When specifying an existing business object, if the name you give is not found, an error message will result. If an error occurs, use the Temporary Query command with wildcards to perform a quick search. For example, to find an object with a name beginning with the letters "HC" and unknown type and revision level, you could enter:

```
temporary query businessobject * HC* *
```

Use: the first `*` for the unknown type, the `HC*` for the name beginning with "HC", and the third `*` for the unknown revision level. The result would be all the objects beginning with "HC".

```
Product HC-430 A

Product HC-470 B
```

Business Object Revision Designator

The third element in a business object specification is the revision label or designator. The revision must be specified if the object requires the revision label in order to distinguish it from other objects with the same name and type. Depending on the object's policy, revisions may or may not be allowed. If they are not allowed or a revision designator does not exist, you must specify `""` (a set of double quotes) for MQL.

The ability (access privilege) to create revisions can be granted or denied depending on the object's state and the session context. When an object is revised, the revision label changes. This label is either manually assigned at the time the revision is created or automatically assigned if a revision sequence has been defined in the governing policy.

Revision sequences provide an easy and reliable way to keep track of object revisions. If the revision sequencing rules call for alphabetic labels, a revised object might have a label such as B or DD. If the Sequence clause in the policy definition specifies custom revision labels, you might see a label such as Unrevised, "1st Rev," "2nd Rev," and so on. In any case, the revision label you provide must agree with the revision sequencing rules. If it does not, an error message will result.

For example, the following are all valid business object specifications:

```
Component "NCR 1139" ""
Drawing "Front Elevation" 2
Recipe "Spud's Fine Mashed Potatoes" IV
"Tax Record" "Sherry Isler" "second rev"
```

The first specification has no revision designator and must be specified as such. This might be because Component types cannot be revised under the governing policy. It might also be because this is the original object that uses a sequence where the first object has no designator.

For more information, see *MQL Command Reference: policy Command: Add Policy: Sequence Clause* in the online user assistance.

Object ID

When business objects are created, they are given an internal ID. As an alternative to `TYPE NAME REV`, you can use this ID when indicating the business object to be acted upon. The ID of an object can be obtained by using the print businessobject selectable "ID". Refer to *Viewing Business Object Definitions* later in this chapter for more information on select commands.

Defining a Business Object

Business objects are defined using the Add Businessobject command:

```
add businessobject BO_NAME policy POLICY_NAME [ITEM {ITEM}];
```

`BO_NAME` is the Type Name Revision of the business object.

`POLICY_NAME` is the policy that governs the business object. The Policy clause is required when creating a new business object. For more details on policies, see *Policies*.

`ITEM` is an Add Businessobject clause that provides additional information about the business object you are creating.

For more information, see *MQL Command Reference: businessobject Command* in the online user assistance.

Working with Legacy Data

When migrating data from a legacy system, you need to create Business objects that represent the actual state of the data you are importing from the legacy system. This is necessary to maintain data integrity. For example, when migrating a document that has been approved in the legacy system you should set the state that correctly indicates its approved state. The business object you create

should also have the actual date/time of the creation and/or modification of the legacy data, and not the system generated date and time (that would reflect when the data was migrated).

In some cases, you may want the business objects to act as a placeholder, representing the actual objects in the external system that is constantly changing. You can update the business objects on a regular basis by resetting the current state as well as the start, end and duration values of the states. Modifying a business object this way avoids the need for promoting the object through the lifecycle, which would result in each state reflecting the date of the import/promotion and not the date from the legacy system. Each of these (current state, start date, end date, and duration) can be changed independently, with no effect on each other. If you want to maintain these values to be consistent with each other, you must change them all. There is no impact on the lifecycle of these objects if these values are not kept consistent, except for queries or webreports that depend on these values.

For more information, see *MQL Command Reference: businessobject Command: Add Business Object: Adding Legacy Data* in the online user assistance.

Viewing Business Object Definitions

You can view the definition of a business object at any time by using the Print Businessobject command and the Select Businessobject command. These commands enable you to view all the files and information used to define the business object. The system attempts to produce output for each select clause input, even if the object does not have a value for it. If this is the case, an empty field is output.

Reserving Business Objects for Updates

When two users try to edit a business object at the same time, the modifications made by one user can overwrite the changes made by another. The same is true for business object connections. To prevent such concurrent modifications, the kernel provides the ability to mark a business object or connection as reserved and store its reservation data.

The kernel does not in itself prevent concurrent modifications. Applications using the kernel can use the reservation data to implement ways to warn or disallow users from modifying a reserved object.

When an object or connection is reserved, the kernel adds the “reserved” tag which includes a user and a timestamp. An application can be programmed such that it checks for the reserved status of a business object or connection, and if reserved, can issue a warning or disallow other users from modifying the business object or connection until it is unreserved.

Implementing reservations in an application

An application can be programmed such that no one can modify a reserved business object or connection unless it is unreserved. To query the reserved status of a business object or connection, implementors can use the following selectables:

Selectable	Description	Output
reserved	Boolean indication reserved status of a business object	True/False
reservedby	Non-empty string or the context user	Name of the person who reserved the object.
reservedstart	The date or timestamp of when the business object was reserved.	Returns the date and time of when that object was reserved.
reservedcomment	Optional comment not exceeding 254 characters from the person reserving the object.	Any comments entered.

For example:

```
print businessobject “Box Design” “Thomas” “A” select reserved
reservedby reservedstart reservedcomment;
```

The above command returns output similar to:

```
reserved = TRUE
reservedby = Jerry
reservedstart = 30-Oct-2005 10:34:10 AM
reservedcomment = “reserved from Create Part Dialog”
```

To query the reserved status of a connection use:

```
print connection 62104.18481.31626.56858 select reserved;
```

Revising Existing Business Objects

The ability to create revisions can be granted or denied depending on the object’s state and the session context. For example, let’s assume you are an editor working on a cookbook. A cookbook is composed of Recipe objects. Recipe objects are created by the Chef who writes the recipe, perhaps in the description field of the object. He then promotes the Recipe to the “Test” state, where he makes the dish and tastes it. At this point, he either approves it and sends it to the next state (perhaps “Submitted for Cookbook”), reassigning ownership to you (the editor), or he may want to revise it, incorporating different ingredients or varying amounts. Therefore, the “Test” state would have to allow revisions by the owner. The Recipe object could then be revised, the new revision starting in the first state of the lifecycle. Once the Recipe is approved, revisions should not be allowed; so, the “Submitted for Cookbook” state would not allow revisions.

In order to maintain revision history, the new object should be created with the revised business object command even though it is possible to create what looks like a new revision by manually assigning a revision designator with the Add or Copy Businessobject command. (However, you can add existing objects to a revision chain, if necessary. Refer to *Adding a Business Object to a Revision Sequence* for more information.) The table below shows the differences between using the

`revise businessobject` and `copy businessobject` commands. For more information, see *MQL Command Reference: businessobject Command* in the online user assistance.

Differences/Similarities between Clones and Revisions		
Property	**Copy or Clone**	**Revision**
Attributes	Values are initially the same but can be modified as part of the clone command.	Values are initially the same but can be modified as part of the revise command.
Files	Files can optionally be copied to the Clone upon creation when the copy command specifies to do so. For more information, see *MQL Command Reference: businessobject Command: Copy Business Object: Handling Files* in the online user assistance.	Files are referenced from the original until it is necessary to copy them when the revise command specifies to do so. For more information, see *MQL Command Reference: businessobject Command: Copy Business Object: Handling Files* in the online user assistance.
Connections to other objects	Depends on the Clone Rules (Float, Replicate, None) set by the Business Administrator.	Depends on the Revision Rules (Float, Replicate, None) set by the Business Administrator.
Connection to Original	No implicit connection.	Implicit connection can be viewed in Revision chain.
History	The create entry shows the object from which it was cloned. If you copy an object via MQL, you can optionally include the original object's history log.	The create entry shows object from which it was revised and original shows that it was revised.

Not all business objects can be revised. If revisions are allowed, the Policy definition may specify the scheme for labeling revisions. This scheme can include letters, numbers, or enumerated values. Therefore you can have revisions with labels such as AA, 31, or "1st Rev."

If the `REVISION_NAME` is omitted, 3DSpace automatically assigns a designator based on the next value in the sequence specified in the object's policy. If there is no sequence defined, an error message results.

If there is no defined revision sequence or if you decide to skip one or more of a sequence's designators, you can manually specify the desired designator as the `REVISION_NAME`.

Adding a Business Object to a Revision Sequence

You can use MQL to add an object to the end of an existing revision sequence using the syntax `revise bus TNR bus TNR1`.

- The first object (`TNR`) must be the last in its revision family. If not, the command will error with `Business object is not revisionable`. Therefore, you cannot insert a revision into the middle of an existing revision family.
- If the second object (`TNR1`) belongs to a revision family, the command will work, but will issue the warning `Business object being inserted will be removed from its previous revision chain`.

Creating new revisions has several side affects. When appending to a revision sequence with the MQL command, these behaviors are handled as described below:

- **Float/Replicate rules**. Ordinarily the float/replicate rules for relationships cause relationships to be moved/copied to new revisions. These rules are ignored; no relationships are added or removed to/from the inserted object, nor are any relationships added or removed to/from any of the previously existing members of the target sequence.
- **File Inheritance**. Ordinarily, a new revision of an existing object inherits all files checked into the original object. When using this interface, if the appended object already has checked in files, it does not inherit any files from the revision sequence. If the appended object has no files checked in, the behavior is controlled by the `file` keyword in MQL.
- **Triggers**. No triggers will fire.
- **History**. Revision history records will be recorded on both the new object and its previous revision (that is, both objects that are specified in the MQL command).

Major/Minor Revisions

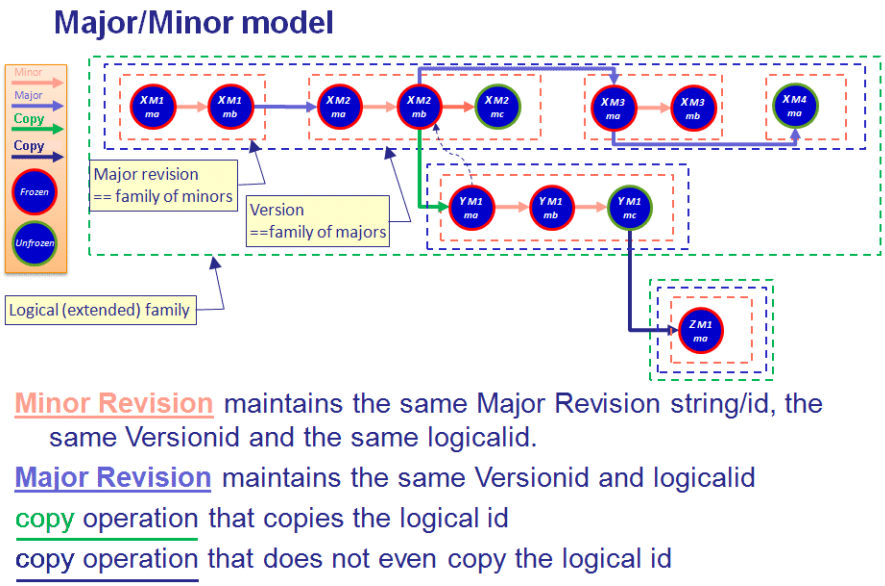
Business objects have two levels of revisioning, minor and major:

- Minor revisions are individual business objects. The Revise Minor command creates a minor revision family as a linear, ordered sequence of objects with a minor revision string that is incremented for each new minor revision.
- Major revisions are equivalent to complete minor revisions. The Revise Major command begins a new major revision (minor revision family) by creating the first minor revision in the new family. An incremented major revision string is generated for each new major revision. You can create as many new major revisions as you want by invoking Revise Major multiple times.

A new UUID called "majorid" is assigned to all objects in a major revision (all objects in the same minor revision family).

The collection of all objects created from an original object using the Revise Major/Minor commands comprises a larger family called a Version Family. A new UUID called VersionId is assigned to all objects in the version family.

The following figure shows an example of the result of a sequence of major and minor revision operations on an object. The orange arrows indicate Revise Minor operations, while the blue arrows indicate Revise Major operations.



The top row of objects shows a version family (enclosed in the dark blue dashed box) comprising four major revisions (i.e., minor revision families enclosed in orange dashed boxes). The four major revisions contain two, three, two, and one minor revisions, respectively.

The second row shows another version family with a single major revision containing three minor revisions. This version family was created through "evolution," or by a Copy operation that copies the logical ID.

The third row shows another version family with a single major revision containing one minor revision. This family was created through a New From operation that did not copy the logical ID.

Validation of MajorID/VersionID during Object Creation

Error checking of majorid/versionid is performed when a business object is added. More specifically, when performing create/clone business object operations (e.g., with the Add Bus command), if the specified majorid or versionid is already associated with other objects in the database, an error is generated. This applies only to Add Bus, not Copy or Revise.

```
add businessobject TYPE NAME REVISION policy POLICY_NAME [ITEM {ITEM}];
…
<< std::endl << " where ITEM is:"
<< std::endl << " | description VALUE |"
…
<< std::endl << " | physicalid UUID |"
<< std::endl << " | logicalid UUID |"
<< std::endl << " | versionid UUID majorid UUID |"
<< std::endl << " | majororder VALUE |"
<< std::endl << " | minororder VALUE |"
…
```

Making Connections Between Business Objects

As described in *Relationships* in Chapter 2, relationship types are created which can be used to link business objects. A *relationship type* is specified when making a *connection* or an instance of that relationship type between two business objects. One business object is labeled as the TO end and one is labeled as the FROM end. When the objects are equivalent, it does not matter which object is assigned to which end. However, in hierarchical relationships, it does matter. 3DSpace uses the TO and FROM labels to determine the direction of the relationship.

The direction you select makes a difference when you examine or dissolve connections. When you examine an object's connections, you can specify whether or not you want to see objects that lead to or away from the chosen object. When you disconnect objects, you must know which object belongs where. Therefore, you should always refer to the relationship definition when working with connections.

Connections are also used to make associations between files, folders, and modules in DesignSync and business objects in Collaboration and Approvals, if you are using DesignSync stores for source control. These types of connections are generally made in apps which use MQL commands in implementation. For more information, see the *Business Process Services - Common User Guide: About DesignSync File Access*.

For more information about the commands that make and break connections between business objects, see *MQL Command Reference: businessobject Command* in the online user assistance. For more information about other commands that can be used to make connections, see *Relationships* in Chapter 2.

Preserving Modification Dates

By default when connections are created or deleted (with connect bus or disconnect bus commands), the modification dates of the objects on both ends of the connection are updated, during which time they are locked. You can use the preserve option on both the mql connect and disconnect commands to avoid this update and the locking of the business objects.

Working with Saved Structures

Once you have saved a structure using the `structure` clause of the Expand Businessobject command, you can list, print, delete, and save it to another user's workspace using the following commands.

```
list structure;

print structure NAME;

delete structure NAME;

copy structure SRC_NAME DST_NAME [fromuser USER_NAME] [touser
USER_NAME][visible USER_NAME{,USER_NAME}] [overwrite];
```

The `print structure` command displays the results in the same manner as `expand bus` does. For example, after executing the above command (which outputs the data to the MQL

window, as well as saving it as a structure), you could execute the following to generate the output again:

```
print structure “Assigned Parts”;
```

Within the print structure command you can also use select clauses on either the business object or the relationship as well as use the output/dump or terse clauses.

The `copy structure` command lets you copy structures to and from any kind of user. Including `overwrite` will replace the copied structure with any structure of the same name that was in the to user’s workspace.

If an object has been disconnected or deleted, it is no longer listed as part of the structure. On the other hand, if other objects were connected since the structure was saved, they would not automatically appear in the output of the print structure command. Another expand command would need to be executed with the structure clause to update the structure.

For more information, see *MQL Command Reference: businessobject Command: Expand Business Object: Structure Clause* in the online user assistance.

Working with Business Object Files

A business object does not need to have files associated with it. It is possible to have business objects where the object's attributes alone are all that is required or desired. However, there will be many cases where you will want to associate external files with a business object. To make this association, you must check the files into the object. This is called file *checkin*.

Checking in a file allows other users to access a file that might not otherwise be accessible. For example, assume you have a file in your personal directory. You would like to make this file accessible to your local group and the quality assurance group. In a typical computer environment, there is no way to allow selective groups to have access while denying others. You could give the file Group Access or World Access. The Group Access takes care of your immediate group but not the quality assurance group. If you give World Access to the file, ANYONE can access it, not just quality assurance. You can overcome this problem with 3DSpace.

The policy definition can designate when specified persons, groups, and roles have access to an object. When an object is accessible, any files that are checked into that object are also accessible. Therefore, if a group has read access to an object, they also have read access to any files checked into the object. If the policy definition for an object includes enforced locking, no checkin for that object is allowed until the lock is released, regardless if the file being checked in is going to replace the checked-out file which initiated the lock or not.

When working with checked in files, keep in mind that the copy you check in will not change if you edit your own personal copy. While you maintain the original, any edits that you make to that file will not automatically appear in 3DSpace. The only way to have those changes visible to other users is to either check in the new version or to make the edits while you are using the 3DEXPERIENCE Platform (i.e., with Open for Edit).

Checking in files is controlled by the Checkin Businessobject command. For more information, see *MQL Command Reference: businessobject Command* in the online user assistance.

Checking Out Files

Once a file is checked in, it can be modified by other users who have editing access (if the object is not locked). This means that the original file you checked in could undergo dramatic changes. As the file is modified, you may want to replace your original copy with one from 3DSpace, or you may want to edit the file externally. This is done by *checking out* the file.

When a business object file is checked out, a copy is made and placed in the location specified. This copy does not affect the Collaboration and Approvals copy. That file is still available to other users. However now you have your own personal copy to work on.

In some situations, a person may be denied editing access but allowed checkout privilege. This means the user may not be allowed to modify the Collaboration and Approvals copy, but can obtain a personal copy for editing. This ensures that the original copy remains intact. For example, a fax template is checked in but each user can check out the template file, fill in individual information, and fax it.

Handling Large Files

The 3DEXPERIENCE Platform handles the transfer of large files (that is, files larger than 2 gb) for checkin or checkout in exactly the same way they handle smaller files. However, the larger a file is, the longer it takes to check it in.

For HTML/JSP-based applications, including the 3DEXPERIENCE apps and custom Framework programs, large file checkins require both of the following:

- Checkins must be targeted for a FCS-enabled store/location (see *Installing File Collaboration Server* for setup information.)
- Checkins must be invoked via the configurable file upload applet (see *Common Components User Guide* for setup information.)

It is recommended that you enable both FCS and the file upload applet for all implementations, even if you do not foresee working with files larger than 2 gb.

Locking and Unlocking a Business Object

A business object can be locked to prevent other users from editing the files it contains. Even if the policy allows the other user to edit it, a lock prevents file edit access to everyone except the person who applied the lock.

Locking a business object protects the object's contents during editing. When an object is opened for editing, it is automatically locked by Collaboration and Approvals, preventing other users from checking in or deleting files. However, if they have the proper access privileges, other users can view and edit attribute values and connections of the object and change its state.

A lock on an object prohibits access to the files it contains, but still allows the object to be manipulated in other ways.

But, if the checkout and edit are separate actions, the object should be manually locked. Without a lock, two people might change an object's file at the same time. With a lock, only the person who locked the object manually is allowed to check files into the object. As with an automatic lock, other users can view attributes, navigate the relationships, and even checkout an object's file. But, they cannot change the contents by checking in or deleting a file without unlocking the object.

It is possible for a locked object to be unlocked by a user with unlock access. For example, a manager may need to unlock an object locked by an employee who is out sick. For more information, see "Studio Modeling Platform Overview: Controlling Access: User Access" in the online user assistance.

If another user has unlock privileges and decides to take advantage of them, the person who established the lock will be notified via IconMail that the lock has been removed and by which user. This should alert the lock originator to check with the *unlocker* (or the history file) before checking the file back in to be sure that another version of the same file (with the same name and format) has not been checked in, potentially losing all edits made by the unlocker.

If the object is governed by a policy which uses enforced locking, the object must be locked for files to be checked in. Users must remember to lock an object upon checkout if they intend to checkin changes, since the separate lock command will be disabled when locking is enforced.

Modifying the State of a Business Object

The following commands control the movement of a business object into or out of a particular state. A state defines a portion of an object's lifecycle. Depending on which state an object is in, a person, group, or role may or may not have access to the object. In some situations, a group should have access but is prohibited because the object has not been promoted into the next state.

Approve Business Object Command

The Approve Businessobject command provides a required signature. When a state is defined within a policy, a signature can be required for the object to be approved and promoted into the next state. You provide the signature with the Approve Businessobject command. For example, to approve an object containing an application for a bank loan, you might write this Approve Businessobject command:

```
approve businessobject "Car Loan" "Ken Brown" A
   signature "Loan Accepted"
   comment "Approved up to a maximum amount of $20,000";
```

In addition to providing the approving signature, a comment was added to provide additional information regarding the approval in the example above. In this case, the comment informs other users that the object (Ken Brown's car loan) has been approved up to an amount of $20,000. If the customer asks for more, the approval would no longer apply and the bank manager might reject it.

The Approve Businessobject command provides a single approving signature. However, the Approve Businessobject signature does not necessarily mean that the object will be promoted to the next state. It only means that one of the requirements for promotion was addressed. Depending on the state definition, more than one signature may be required.

Ignore Business Object Command

The Ignore Businessobject command bypasses a required signature. In this case, you are not providing an approving or rejecting signature. Instead you are specifying that this required signature can be ignored for this object.

In a policy definition, states are created to serve the majority of business objects of a particular type. This means that you may have some business objects that do not need to adhere to all of the constraints of the policy. For example, you might have a policy for developing software programs. Under this policy, you may have objects that contain programs for customer use and programs for internal use only. In the case of the internal programs, you may not want to require all of the signatures for external programs. Instead, you might be willing to ignore selected signatures since the programs are not of enough importance to warrant them.

Use the Ignore Businessobject command to bypass a required signature. For example, assume you have a simple inventory program to track one group's supplies. The program is highly specialized for internal use and will not be used outside the company. According to the policy governing the object (which contains the program), the company president's signature is required before the program can enter the Released state for business objects outside the company. Since no one wants to bother the president for a signature, you decide to bypass the signature requirement with the following Ignore Businessobject command. (Note that the user must have privileges to do this.)

```
ignore businessobject "Software Program" "In-house Inventory" III
   signature "Full Release"
   comment "Signature is ignored since program is for internal use only";
```

In this command, the reason for the bypass is clearly defined so that users understand the reason for the initial bypass of the signature. As time passes, this information could easily become lost. For that reason, you should include a Comment clause in the command even though it is optional.

The Ignore Businessobject command involves control over a single signature. An object is promoted to the next state automatically when all requirements are met if the state was defined with the Promote clause set to true. Bypassing the Ignore Businessobject signature does not necessarily mean that the object will meet all of the requirements for promotion to the next state. It only means that one of the requirements for promotion was circumvented. Depending on the state definition, another user or condition may be required to approve the program.

Reject Business Object Command

The Reject Businessobject command provides a required signature. In this case, the signature is used to prevent an object from being promoted to the next state.

For example, a bank manager may decide that more clarification is required before s/he will approve of car loan. S/he can enter this information by writing the following Reject Businessobject command:

```
reject businessobject “Car Loan” “Ken Brown” A
   signature “Loan Accepted”
   comment “Need verification of payoff on student loan before I’ll approve
      a loan up to a maximum amount of $20,000”;
```

Any other users can see the reason for the rejection. Since the signature was provided, the object cannot be promoted unless someone else overrides the signature or the reason for rejection is addressed.

The Reject Businessobject command provides a single signature. However, this signature does not necessarily mean that the object will be demoted or completely prevented from promotion to the next state. It only means that one of the requirements for promotion was denied. Depending on the state definition, another user may override the rejection.

Unsign Signature

The Unsign Signature command is used to erase signatures in the current state of an object. Any user with access to approve, reject, or ignore the signature has access to unsign the signature.

For example the command to unsign the signature "Complete" in object Engineering Order 000234 1 is:

```
unsign businessobject "Engineering Order" 000234 1 signature Complete;
```

Errors will occur under the following conditions:

- Attempts to unsign a signature not yet signed.
- Attempts to unsign all signatures if all are not signed, any that are signed; however, will be unsigned.
- Attempts to unsign a non-existent signature.
- Attempts to unsign a signature without access.

Disable Business Object Command

The Disable Businessobject command holds an object in a particular state indefinitely. When a business object is in a disabled state, it cannot be promoted or demoted from that state. Even if all

of the requirements for promotion or demotion are met, the object cannot change its state until the state is enabled again.

Use the Disable Businessobject command to disable a business object within a state:

```
disable businessobject OBJECTID [state STATE_NAME];
```

`OBJECTID` is the `OID` or Type Name Revision of the business object.

`STATE_NAME` is the name of the state in which you want to freeze the object. If you want to disable the current state, the State clause is not required.

For example, to disable an object containing a component design for an assembly, you could write a Disable Businessobject command as:

```
disable businessobject “Component Design” “Bicycle Seat R21” A
   state “Initial Release”;
```

Labeling an object as disabled within a state does not affect the current access. This means that the designer can continue to work on the object in that state.

Enable Business Object Command

The Enable Businessobject command reinstates movement of an object. When a business object is in a disabled state, it cannot be promoted or demoted from that state. If you then decide to allow an object to be promoted or demoted, you must re-enable it using the Enable Businessobject command.

When an object is first created, all states are enabled for that object. If a manager or other user with the required authority decides that some states should be disabled, s/he can prevent promotion with the Disable Businessobject command. After the command is processed, an object will remain trapped within that state until it is enabled again.

Use the Enable Businessobject command to enable a business object within a state:

```
enable businessobject OBJECTID [state STATE_NAME];
```

`OBJECTID` is the `OID` or Type Name Revision of the business object.

`STATE_NAME` is the name of the state in which you want to enable the object. If you want to enable the current state, the State clause is not required.

For example, assume Component Design is disabled. All of the conditions for an Initial Release state have been met and the manager decides it is ready for promotion into the Testing state. Before the object can be promoted, however, it must be enabled. The manager can do this with the following command:

```
enable businessobject “Component Design” “Bicycle Seat R21” A
 state “Initial Release”;
```

The object is enabled and available for promotion or demotion.

Enabling an object does not have an effect on the remaining states. For example, the Testing state could be disabled at the same time the Initial Release state was disabled. When the Initial Release state is enabled, the object can be promoted into the Testing state. However, once it is in this new state, it again can no longer be promoted or demoted. It will remain in the Testing state until it is enabled for that state.

Override Business Object Command

The Override Businessobject command turns off the signature requirements and lets you promote the object. Since a policy is a generic outline that addresses the majority of needs, it is possible to have special circumstances in which a user, such as a manager, may decide to skip a state rather than have the object enter the state and try to meet the requirements of the state. This is done using the Override Businessobject command.

Use The following syntax to write an Override Businessobject command:

```
override businessobject OBJECTID [state STATE_NAME];
```

`OBJECTID` is the `OID` or Type Name Revision of the business object.

`STATE_NAME` is the name of the state that you want to skip. If you want to override the current state, the State clause is not required.

For example, assume you have a component that has undergone cosmetic changes only. This component is needed in Final Release although the policy dictates that the component must go through Testing before it can enter that state. Since the changes were cosmetic only, the manager may decide to override the Testing state so that the users can access the component sooner. This could be done by entering a command similar to:

```
override businessobject “Component Design” “Bicycle Seat R21”
   state Testing;
```

Now, assume that the object is currently in the Initial Release state and a promotion would place it in the Testing state. How does the Override Businessobject command affect the object when it is promoted? The object will be promoted directly from the Initial Release state into the Final Release state.

Promote Business Object Command

The Promote Businessobject command moves an object from its current state into the next state. If the policy specifies only one state, an object cannot be promoted. However, if a policy has several states, promotion is the means of transferring an object from one state into the next.

The order in which states are defined is the order in which an object will move when it is promoted. If all of the requirements for a particular state are met, the object can change its state with the Promote Businessobject command.

```
promote businessobject OBJECTID;
```

`OBJECTID` is the `OID` or Type Name Revision of the business object.

For example, the following command would promote an object containing an application for a bank loan:

```
promote businessobject “Car Loan” “Ken Brown” A;
```

An object cannot be promoted if:

- Its current state is disabled.
- The signature requirements were not met or overridden (ignored).
- There are no other states beyond the current state.

Demote Business Object Command

The Demote Businessobject command moves an object from its current state back into its previous state. If the policy has only one state or is in the first state, an object cannot be demoted. However, if a policy has several states, demotion is the means of transferring an object backward from one state into the previous state.

If an object reaches a state and you determine that it is not ready for that state, you would want to send the object back for more work. This is the function of the Demote Businessobject command:

```
demote businessobject OBJECTID;
```

`OBJECTID` is the `OID` or Type Name Revision of the business object.

For example, assume you have a component that has undergone cosmetic changes only. The manager decides to send it into Final Release without sending it through testing. Now the manager finds out that the new paint trim might weaken the plastic used in the seat. Therefore, the manager decides to demote the object back into the Testing state. This could be done by entering a command similar to:

```
demote businessobject “Component Design” “Bicycle Seat R21” A;
```

When using the Demote Businessobject command, an object cannot be demoted if:

- The current state is disabled.
- The signature requirements for rejection were not met or overridden (ignored).
- There are no other states prior to the current state.

Working with Relationship Instances

Once you have defined relationship types, connections or instances of a relationship type can be made between specific business objects. Refer to *Making Connections Between Business Objects* for the MQL commands: Connect Businessobject and Disconnect Businessobject. Other commands that can be used to make connections and work with connections in general are described in the following section.

Connections are also used to make associations between files, folders and modules in DesignSync and business objects in Collaboration and Approvals, if you are using DesignSync stores for source control. These types of connections are generally made in 3DEXPERIENCE apps, which use MQL commands in implementation. For more information, see the *Common Components User Guide: About DesignSync File Access*.

Connections can be accessed by MQL in two ways:

- by specifying the business objects on each end

Or:

- by specifying its connection ID.

The first method will work only if exactly one of a particular relationship type exists between the two listed objects. Therefore, if connections are to be programmatically modified, the second method, using connection IDs, is the safer approach.

Defining a Connection

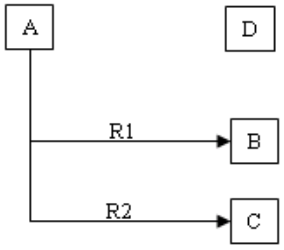
There are two ways of creating a new instance of a relationship: the Connect Businessobject command or the Add Connection command. Use the Add Connection command to make various types of connections between either:

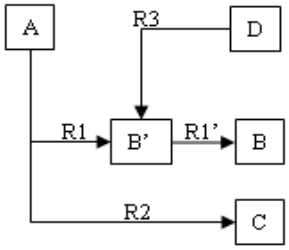
- 2 business objects
- a business object and a connection
- 2 connections

One business object or connection is labeled as the TO or TOREL end and one is labeled as the FROM or FROMREL end. When the objects are equivalent, it does not matter which object is assigned to which end. However, in hierarchical relationships, it does matter. Collaboration and Approvals use the TO and FROM labels to determine the direction of the relationship.

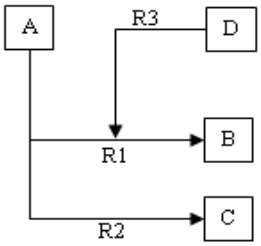
The direction you select makes a difference when you examine or dissolve connections. When you examine an object's connections, you can specify whether or not you want to see objects that lead to or away from the chosen object. When you disconnect objects, you must know which object belongs where. Therefore, you should always refer to the relationship definition when working with connections. To disconnect objects, see *Making Connections Between Business Objects*.

Creating connections that have a connection on at least 1 end eliminates the need to create dummy business objects when modeling schema. For example, if a connection can not be used as an end point, to connect object D to the R1 connection in the diagram below, you would have to create an extra business object and an extra connection so that D is logically tied to R1.





Directly connecting the object to the connection is a streamlined modeling approach resulting in reduced storage requirements and enhanced performance levels.



You can connect an object or connection to or from a connection if you have to/from connect access on the object or connection. Likewise, you can delete (disconnect) or modify the connection with appropriate access on the object or connection that is on one end of it.

You cannot use Matrix Navigator to create connections that have a connection on one or both ends. This must be done using MQL or Studio Customization Toolkit programs.

Add Connection Command

Use the Add Connection command to create new connections. You can specify either a business object or a connection at either end of the connection

```
add connection NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the connection.

`ADD_ITEM` is an Add Connection clause which provides more information about the connection you are creating.

For more information, see *MQL Command Reference: connection Command* in the online user assistance.

Viewing Connection Definitions

You can view the definition of a connection at any time by using the Print Connection command and the Select Connection command. These commands enable you to view all the files and information used to define the connection. The system attempts to produce output for each select clause input, even if the connection does not have a value for it. If this is the case, an empty field is output

4 Working with Workspace Objects

This chapter discusses all the different types workspace objects.

In this section:

Queries

A *query* is a search on the database for objects that meet the specified criteria. The query is formulated by an individual and, in MQL, it must be saved for subsequent evaluation. A user has access only to queries created during a session under her or his own context. It is then run or *evaluated* and 3DSpace finds the objects that fit the query specification. The found objects are displayed in Collaboration and Approvals or listed in MQL. If the found objects are often needed as a group, they can be saved in a set which can be loaded at any time during an Collaboration and Approvals or MQL session under the same context.

There are two steps to working with queries:

- *Define* either a saved query that you want to use again, or a temporary query to be used only once for a quick search.
- *Evaluate* the query. The query is processed and any found objects can be put into a set.

Temporary queries allow you to perform a quick search for objects you need only once. In this case you don't have to first save the query itself as an object. For example, you might want to modify a particular object that is named HC-4....., but you have forgotten its full name or capitalization. You could perform a temporary search (without saving the actual query) using "HC-4*" to find all objects that have a name beginning with the letters "HC-4". From the resulting list, you could enter the correct name in your modify command.

Saved queries allow you to find, for example, all drawings created for a particular project. You can save the results of the query in a set. As the project proceeds, you can also name and save the query itself to use again to update the set contents. Or you might want to repeatedly search for any objects having a particular attribute or range of attributes. Saved queries provide a way of finding all the objects that contain the desired information.

Defining a Query

To define a saved query from within MQL, use the Add Query command:

```
add query NAME [user USER_NAME] {ITEM};
```

`NAME` is the name you assign to the query.

`USER_NAME` can be included with the user keyword if you are a business administrator with person access defining a query for another user. If not specified, the query is part of the current user's workspace.

`ITEM` specifies the characteristics to search for.

For more information, see *MQL Command Reference: query Command* in the online user assistance.

Sets

A *set* is a logical grouping of business objects created by an individual user. Sets are often the result of a query made by a user. For example, the user may want to view all drawings created for a particular project or find information about objects having a particular attribute or range of attributes. While sets can be manually constructed based on the needs and desires of the individual, queries are a fast means of finding related business objects.

The contents of a set can be viewed at any time and objects can be added or deleted from a set easily. However, a user has access only to sets created while in session under his/her context.

Understanding Differences

It is important to realize that sets are not the same as connections between business objects. Connections also group business objects together in a window for users to analyze; however, connections are globally known links rather than local links.

Business Object Connection	Set
Created only if the policy permits.	Created without any special privileges.
Available to view by all users who have access to the objects.	Available to view at any time by the person who created the set.
Valuable to all users who have access to the objects.	Valuable only within the context of the person who created it.

Defining a Set

Sets are often the result of a query. To save the contents of a query in a set, see *MQL Command Reference: query Command: Evaluate Query* in the online user assistance.

To manually define a set from within MQL, use the Add Set command:

```
add set NAME [user USER_NAME][ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the set.

`USER_NAME` can be included with the user keyword if you are a business administrator with person access defining a set for another user. If not specified, the set is part of the current user's workspace.

`ADD_ITEM` is an Add Set clause that provides more information about the set you are creating.

For more information, see *MQL Command Reference: set Command* in the online user assistance.

Filters

Filters limit the objects or relationships displayed in browsers to those that meet certain conditions previously set by you or your Business Administrator. For example, you could create a filter that would display only objects in a certain state (such as Active), and only the relationships connected *toward* each object (not *to and from*). When this filter is turned on, only the objects you needed to perform a specific task would display.

From Matrix Navigator browsers, filters that limit the number of objects that display can be very useful. Each user can create her/his own filters from Matrix Navigator (or MQL). However, if your organization wants all users to use a basic set of similar filters consistently, it may be easier to create them in MQL, then copy the code to each user's personal settings (by setting context).

Filters can be defined and managed from within MQL or from the Visuals Manager in Matrix Navigator. Once the filters are defined, individual users can turn them on and off from the browsers as they are needed.

In the Matrix Navigator browsers, filters display on the Filters tab page within the Visuals Manager window, in the Filter bar and in the View menu.

It is important to note that filters are Personal Settings that are available and activated only when context is set to the person who defined them.

Defining Filters

To define a new filter from within MQL, use the Add Filter command:

```
add filter NAME [user USER_NAME] [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the filter.

`USER_NAME` can be included with the user keyword if you are a business administrator with person access defining a filter for another user. If not specified, the cue is part of the current user's workspace.

`ADD_ITEM` specifies the characteristics you are setting.

For more information, see *MQL Command Reference: filter Command* in the online user assistance.

Cues

Cues control the appearance of business objects and relationships inside any browser. They make certain objects and relationships stand out visually for the user who created them.

This appearance control is based on conditions that you specify such as attribute value, current state, or lateness. Objects and relationships that meet the criteria may appear in any distinct color, line, or font style.

You can save a cue and not make it active. This allows for multiple or different sets of conditions to be used at different times or as a part of different Views. See “Using View Manager” in the *Matrix Navigator Guide*. Cues are set using the Visuals Manager and saved as a query in your personal settings. They can be activated from the Visuals Manager Cue tab or from the View menu.

From Matrix Navigator browsers, cues that highlight the appearance of certain objects can be very useful. Each user can create her/his own cues from Matrix Navigator (or MQL). However, if your organization wants all users to use a basic set of similar cues consistently, it may be easier to create them in MQL, then copy the code to each user’s personal settings (by setting context).

Cues can be defined and managed from within MQL or from the Visuals Manager in Matrix Navigator. Once the cues are defined, individual users can turn them on and off from the browsers as they are needed.

In the Matrix Navigator browsers, they display on the Cues tab page within the Visuals Manager window and in the View menu.

It is important to note that cues are all Personal Settings that are available and activated only when context is set to the person who defined them.

Defining a Cue

From browsers, unique cues can be very useful when interacting with many objects. Each user can create her/his own cues from Matrix Navigator (or MQL). However, if your organization wants all users to see a basic set of similar cues consistently, it may be easier to create them in MQL, then copy the code to each user’s personal settings (by setting context).

To create a new cue from within MQL, use the Add Cue command:

```
add cue NAME [user USER_NAME] [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the nameyou assign to the cue.

`USER_NAME` can be included with the user keyword if you are a business administrator with person access defining a cue for another user. If not specified, the cue is part of the current user’s workspace.

`ITEM` specifies the characteristics you are setting.

For more information, see *MQL Command Reference: cue Command* in the online user assistance.

Inquiries

Inquiries can be evaluated to produce a list of objects to be loaded into a table in a JSP application. In general, the idea is to produce a list of business object ids, since they are the fastest way of identifying objects for loading into browsers. Inquiries include code, which is generally defined as an MQL `temp query` or `expand bus` command, as well as information on how to parse the returned results into a list of OIDs.

Defining an Inquiry

Business Administrators can create new inquiry objects if they have the Inquiry administrative access. To create a new inquiry from within MQL, use the Add Inquiry command:

```
add inquiry NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the nameyou assign to the inquiry.

`ITEM` specifies the characteristics you are setting.

For more information, see *MQL Command Reference: inquiry Command* in the online user assistance.

5 Extending an Application

This chapter discusses concepts on administrative objects used for extending applications.

Programs

A *program* is an object created by a Business Administrator to execute specific commands. Programs are used:

- In format definitions for the edit, view, and print commands.
- As Action, Check, or Override Event Triggers, or as actions or checks in the lifecycle of a policy.
- To run as *methods* associated with certain object types. (Refer to *Types* in Chapter 2, for the procedure to associate a program with a type.)
- In Business Wizards, both as components of the wizard and to provide the functionality of the wizard.
- To populate attribute ranges with dynamic values.
- In expressions used in access filters, where clauses and configurable tables.

Many programs installed with the Framework include Java code, which are invoked while performing operations with apps. This type of program is defined as *java*. The majority of your programs should be Java programs (JPO), particularly if your users are accessing the 3DEXPERIENCE Platformwith a Web browser.

Some programs might execute operating system commands. This type of program is *external*. Examples are programs such as a word processor or a CAD program which can be specified as the program to be used for the edit, view, and print commands in a format definition.

Other programs might use only MQL/Tcl commands (although this technology is older, and Java programs written with the Studio Customization Toolkit will perform better, particularly in a Web environment.) For example, a check on a state might verify the existence of an object using an MQL program.

Some programs may require a business object as the *context* or starting point of the commands. An example of this is a program that connects a business object to another object.

Java Program Objects

A Java Program Object (JPO) contains code written in the Java language. JPOs provide the ability to run Java programs natively inside the kernel, without creating a separate process and with a true object-oriented integration ‘feel’ as opposed to working in MQL. JPOs allow developers to write Java programs using the Studio Customization Toolkit programming interface and have those programs invoked dynamically.

When running inside the 3DSpace kernel, programs share the same context and transaction as the thread from which they were launched. In fact, Java programs run inside the same Java Virtual Machine (JVM) as the kernel.

JPOs are also tightly integrated with the scripting facilities of the 3DEXPERIENCE Platform. Developers can seamlessly combine MQL, Tcl, and Java to implement their business model. However, while Tcl will continue to be supported and does offer a scripting approach which can makes sense in some cases, the Java language brings several advantages over Tcl:

- Java is compiled, and therefore offers better run-time performance
- Java is object-oriented
- Java is thread safe

Java code must be contained in a class. In general, a single class will make up the code of a JPO. However, simply translating Tcl program objects into Java is not the goal. This would lead to many small classes, each containing a single method. The very object-oriented nature of Java lends itself to encapsulating multiple methods into a single class. The goal is to encapsulate common code into a single class.

A JPO can be written to provide any logical set of utilities. For example, there might be a JPO that provides access to Administration objects that are not available in the Studio Customization Toolkit. But for the most part, a JPO will be associated with a particular Business Type. The name of the JPO should contain the name of the Business Type (but this is certainly not required).

It is the responsibility of the JPO programmer to manually create a JPO and write all of the methods inside the JPO. Keep in mind that JPO code should be considered server-side Java; no Java GUI components should ever be constructed in a JPO.

For more details about writing JPO code, see the *Configuration Guide: About Java Program Objects (JPOs)*.

Defining a Program

A program is created with the Add Program command:

```
add program NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the program.

`ADD_ITEM` is an Add Program clause which provides additional information about the program you are creating.

For more information, see *MQL Command Reference: program Command* in the online user assistance.

Using Programs

For information on using programs in an implementation, see the *Configuration Guide: About Programs*. It contains key information including:

- compiling programs
- extracting and inserting JPO code
- passing arguments
- Java options

The sections that follow include some basic MQL syntax for the above functionality.

Compile command

To force compilation before invoking a JPO method, use `compile program`. This is useful for bulk compiling and testing for compile errors after an iterative change to the JPO source.

```
compile program PATTERN [force] [update];
```

Compiler flags can be adjusted using the `MX_JAVAC_FLAGS` environment variable. For more information, see *Live Collaboration Server: Configuring Live Collaboration: Live Collaboration Settings: Optional Variables*.

When a JPO is compiled or executed, any other JPOs which are called by that JPO or call that JPO must be available in their most recent version. The compile command includes an update option

which will update the requested JPO dependencies on other JPOs that may have been added, deleted, or modified.

Execute command

You can run a program with the execute program command:

```
exec program PROGRAM_NAME [-method METHOD_NAME] [ARGS]
[-construct ARG];
```

where:

ARGS is zero or more space-delimited strings.

The `-construct` clause is used to pass arguments to the constructor. ARG is a single string. If more than one argument needs to be passed to the constructor, the `-construct` clause can be repeated as many times as necessary on a single command.

Extract command

Extracting the Java source in the form of a file out to a directory is useful for working in an IDE. While in the IDE a user can edit, compile, and debug code. The `extract program` command processes any special macros and generates a file containing the Java source of the JPO. If no source directory is given, the system uses ENOVIA_INSTALL/java/custom (which is added to `MX_CLASSPATH` automatically by the install program).

```
extract program PATTERN [source DIRECTORY]
```

In order to use the extract feature, the JPO name must follow the Java language naming convention (i.e., no spaces, special characters, etc.). Only alphanumeric printable characters as well as '.' and '_' are allowed in class names in the JPO.

Insert command

After testing and modifying Java source in an IDE, it is necessary to insert the code back into JPOs in the database. The `insert program` command regenerates special macros in the Java source as it is placed back into a JPO (reverse name-mangling). If the JPO does not exist, the `insert` command creates it automatically.

```
insert program FILENAME | DIRECTORY;
```

For example:

```
insert program matrix/java/custom/testjpo_mxJPO.java
```

OR

```
insert program matrix/java/custom/
```

The later will insert all the .java files in the specified directory.

History

Collaboration and Approvals provides a history for each business object, detailing every activity that has taken place since the object was created.

An object's historical information includes:

- The type of activity performed on the object, such as create, modify, connect, and so on.
- The user who initiated the activity.
- The date and time of the activity.
- The object's state when the activity took place.
- Any attributes applicable when the activity took place.

You can add a customized history through MQL to track certain events, either manually or programmatically.

History records can be selected when printing or expanding business object, set, or connection information using the MQL `select` clause. In addition, when printing all information about a business object, set, or connection, history records can be excluded.

System administrators only can purge the history records of a business object or connection via MQL. In addition, *all* history records of a business object or connection can be deleted with one command.

History can be turned off in a single session by a System Administrator for the duration of the session or until turned back on. In addition, if an implementation does not require history recording, or requires only custom history entries, Collaboration and Approvals "standard" history can be disabled for the entire system. Turning history recording off can improve performance in very large databases, though certain standards may require that it is turned on.

History entries larger than 255 characters are truncated to 255 characters. This includes custom history entries as well as Collaboration and Approvals history entries. This means that history logs for the modification of long description or string attribute fields may be truncated.

When objects are created and then immediately modified within a trigger, the timestamp is often identical. When this happens, the modify event may be logged before the create event, although both will have the same timestamp.

For more information, see *MQL Command Reference: history Command* in the online user assistance.

Administrative Properties

Ad hoc attributes, called Properties, can be assigned to an administrative object by business administrators with Property access. Properties allow links to exist between administrative definitions that aren't already associated. There are two kinds of properties:

- *User properties*, which can be created by users to suit their needs. They can apply to all administrative objects, including workspace objects.
- *System properties*, which come with Collaboration and Approvals. These properties are used internally to implement certain kinds of administrative objects. For example, toolsets point to their programs via a property; views point to their components in the same way.

Properties may be useful for developers who are integrating Collaboration and Approvals to other application programs. However, the typical Business Administrator may never have the need to use properties.

Properties can be created, modified, displayed, and deleted only through MQL. However, MQL can be embedded in programs, `where` clauses and `select` clauses, making properties available to a broader audience.

Defining a Property

Properties can be created and attached to an object at the same time using the `add property` command. A property must have a name and be "on" an object. It can, optionally, define a link to another administrative object using the "to" clause. This command, therefore, takes two forms, with and without the "to" clause.

```
add property NAME on ADMIN_TYPE ADMIN_NAME [system] [to
ADMIN_TYPE ADMIN_NAME [system]] [value VALUE];
```

`NAME` is the name of the new property.

`ADMIN_TYPE` is the keyword for an administrative or workspace object:

association	group	policy	site	view
attribute	index	program	store	wizard
command	inquiry	query	table	
cue	location	relationship	tip	
filter	menu	role	toolset	
form	page	rule	type	
format	person	set	vault	

`ADMIN_NAME` is the name of the administrative object instance.

The `to ADMIN_TYPE ADMIN_NAME` is optional.

`system` is used only when adding properties on/to system tables.

`VALUE` is a string value of the property. The "value" clause is optional. The value string can contain up to 2gb of data.

For more information, see *MQL Command Reference: property Command* in the online user assistance.

Applications

You can design your application's data model such that you can mark part or all of it as private or protected. The data you mark private cannot be accessed from any other project while the data you mark protected can only be viewed and not modified. If your company is working on a top secret government project, you will need to mark the data connected to this project as private to avoid any accidental viewing or modification of data. You can do this by assigning an owning application to your project.

An *application* is a collection of administrative objects (attributes, relationships and types) defined by the Business Administrator and assigned to a project. It acts as a central place where all application dependent associations to these other administration objects are defined. The application members (the types, relationships etc) can have different levels of protection (private, protected or public). This protection also extends to the objects governed by them. For example, if you mark a relationship as protected, all connections of that type will also get marked as protected.

The advantage of assigning an owning application to a project is that it ensures data integrity. All modifications to the data are handled by the owning application code which performs all necessary checks to ensure complete data integrity. This prevents any unintended mishandling and possible corruption of data. Thus, when an Collaboration and Approvals product (or any custom application) tries to access data marked private or protected by connecting to the Server, it has to specify the name of the owning application containing that data. You can specify an application name only in the Studio Customization Toolkit.

In MQL and Business Modeler, a person can be assigned an owning application.

Defining an Application

An application is created with the Add Application command:

```
add application NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the application.

`ADD_ITEM` is an Add Application clause which provides additional information about the application you are creating.

For more information, see *MQL Command Reference: application Command* in the online user assistance.

Dataobjects

Dataobjects are a type of workspace object that provide a storage space for preference settings and other stored values for users. Apps will use them for cached values for form fields, as well as to add personalized pages to channels in PowerViews. Refer to *Channels* in Chapter 6 for more information.

Settings stored in dataobjects are not limited in length.

Defining a Dataobject

Dataobjects can be created in MQL only.

To create a new dataobject, use the Add dataobject command:

```
add dataobject NAME [user USER_NAME] [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the dataobject.

`USER_NAME` can be included with the user keyword if you are a business administrator with person access defining a dataobject for another user. If not specified, the dataobject is part of the current user's workspace.

`ADD_ITEM` further defines the dataobject.

For more information, see *MQL Command Reference: dataobject Command* in the online user assistance.

Webreports

A webreport is a workspace object that can be created and modified in MQL or using the WebReport class in the Studio Customization Toolkit. Webreports are used to obtain a set of statistics about a collection of business objects or connections. The administrative definition of a webreport includes:

- search criteria which specifies the full set of objects to be examined.
- one or more groupby criteria which specify how to organize the objects into groups
- one or more data expressions to be calculated on each group. These are expressions suitable for evaluating against a collection of business objects – such as count, average, maximum, minimum, etc.

Once a webreport is created it can be evaluated to produce a webreport result, which consists of both the organized set of data values and objects for the subgroups. It can be saved to the database (with or without the corresponding business objects) if desired. Webreport results can also be archived and a webreport can store any number of archived results as well as a single result referred to as “the result”. Webreports are used mainly by the Business Metrics component of Business Process Services, but can be used in custom applications as well.

Defining a Webreport

To define a webreport from within MQL use the add webreport command:

```
add webreport NAME [user USER_NAME][ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the webreport.

`ADD_ITEM` is an add webreport clause that provides additional information about the webreport.

For more information, see *MQL Command Reference: webreport Command* in the online user assistance.

Evaluating Webreports

When a webreport is evaluated, it can either report the results back to the caller, save them in the database, or both. When results are saved, so is the following information:

- the date/time evaluated;
- the time it took to evaluate;
- the name of the person who ran it (the context user).

Also when results are saved, they overwrite any previously saved results unless those results have been archived. Any number of results can be archived.

Webreport XML Result

There is a limitation on the size of the XML result string for a webreport on a non-Oracle database. If the XML result string exceeds 2097152 characters, an error may occur.

Since the webreport XML result is summarizing the data values for all of the subgrouping implied by the webreport definition, there are two cases where this XML result can get somewhat lengthy, and these should be avoided on non-Oracle databases:

- If any single groupby expression takes on a very large number of distinct values across the objects in a searchcriteria, such as if a groupby is defined in terms of an unconstrained attribute (no range value definitions).

  Also, groupby=owner could take on 1000’s of values, and groupby=id will result in each object being in its own subgrouping.
- If the webreport has a large number of groupbys. The total number of cells (or subgroupings) in a webreport will be equal to N1 * N2 * Nk, where N1 represents the distinct values the first groupby expression takes on, N2 represents the distinct values the second groupby expression takes on, etc.

6 Modeling Web Elements

This chapter discusses concepts on administrative objects used for modeling web elements.

Commands

Business Administrators can create new command objects if they have the Menu administrative access. Commands can be used in any kind of menu in a JSP application. Commands may or may not contain code, but they always indicate how to generate another Web page. Commands are child objects of menus — commands are created first and then added to menu definitions, similar to the association between types and attributes. Changes made in any definition are instantly available to the applications that use it.

Commands can be role-based, that is, only shown to particular users. For example, a number of commands may only be available when a person is logged in as a user defined as Administrator. When no users are specified in the command definitions, they are globally available to all users.

Defining a Command

To define a command from within MQL use the Add Command command:

```
add command NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the nameyou assign to the command.

You cannot have both a command and a menu with the same name.

`ADD_ITEM` is an Add Command clause that provides additional information about the command.

For more information, see *MQL Command Reference: command Command* in the online user assistance.

Using Macros and Expressions in Configurable Components

Many strings used in the definition of configurable Components (such as label values, hrefs, and settings) can contain embedded macros and select clauses. The ${} delimiters identify macro names. Macros are evaluated at run-time. Macros for configurable components are available for directory specification. Some existing macros are also supported (refer to *Supported Macros and Selects* for more information).

Some strings can also include `select` clauses which are evaluated against the appropriate business object at run-time. The $<> delimiters identify select clauses. Because the select clauses will generally use symbolic names, the clauses will be preprocessed to perform any substitutions before submitting for evaluation. The following example shows a macro being used in the href definition and another macro being used in the Image setting, as well as a select clause being used in the label definition of a tree menu (associated with a LineItem object):

```
MQL<2>print menu type_LineItem;
menu type_LineItem
  description
  label '$<attribute[attribute_EnteredName].value>'
  href '${SUITE_DIR}/emxQuoteLineItemDetailsFS.jsp'
  setting Image value ${COMMON_DIR}/iconSmallRFQLineItem.gif
  setting Registered Suite value SupplierCentralSourcing
  children
    command SCSAttachment
    command SCSAttributeGroup
    command SCSHistory
```

```
    command SCSSupplierExclusion
    command SCSUDA
  nothidden
  property original name value type_LineItem
  property installed date value 02-28-2002
  property installer value MatrixOneEngineering
  property version value Verdi-0-0-0
  property application value Sourcing
  created Thu Feb 28, 2002 11:12:34 AM EST
  modified Thu Feb 28, 2002 11:12:34 AM EST
```

The following example shows a typical business object macro being used in the label definition of a tree menu (associated with a Company object):

```
MQL<3>print menu type_Company;
menu type_Company
  description
  label '${NAME}'
  href '${SUITE_DIR}/emxTeamCompanyDetailsFS.jsp'
  setting Image value ${COMMON_DIR}/iconSmallOrganization.gif
  setting Registered Suite value TeamCentral
  children
    command TMCBusinessUnit
    command TMCLocation
    command TMCPeople
  nothidden
  property original name value type_Company
  property installed date value 02-28-2002
  property installer value MatrixOneEngineering
  property version value Verdi-0-0-0
  property application value TeamCentral
  created Thu Feb 28, 2002 11:31:57 AM EST
  modified Thu Feb 28, 2002 11:31:57 AM EST
```

When using a macro, surround it with quotes to ensure proper substitution if a value contains spaces.

Supported Macros and Selects

The following sections provide lists of macros used in the configuration parameters of the administrative menu and command objects found in the Framework. These menu and command objects are used for configuring the menus/trees in the apps that use them. These are the only macros currently supported for use in any dynamic UI component.

Directory Macros

The following table provides the list of directory specific macros used in the configuration setting.

Directory Macros	
Macro Name	**Description**
${COMMON_DIR}	To substitute the “common” directory below “ematrix” directory. The substitution is done with reference to any application specific directory and it is relative to the current directory.

Directory Macros	
${ROOT_DIR}	To substitute the “ematrix” directory. The substitution is done with reference to any application specific directory below “ematrix” and it is relative to the current directory.
${SUITE_DIR}	The macro to substitute the application specific directory below “ematrix” directory. The substitution is done based on the “Suite” to which the command belongs. and it is relative to the current directory.

Select Expression Macros

Select expression macros are defined as $<SELECT EXPRESSION>, where the select expression can be any valid MQL select command. Select expression macros can be used in labels for configurable components and in expression parameters. These expressions are evaluated at runtime against the current business object ID and relationship ID that is passed in. Some examples include:

- $<TYPE>
- $<NAME>
- $<REVISION>
- $<attribute[attribute_Originator].value>
- $<attribute[FindNumber].value>
- $<from[relationship_EBOM].to.name>

Menus

Business Administrators can create new menu objects if they have the Menu administrative access. Menus can be used in custom Java applications. Before creating a menu, you must define the commands that it will contain, since commands are child objects of menus — commands are created first and then added to menu definitions, similar to the association between types and attributes. Changes made in any definition are instantly available to the applications that use it. Menus can be designed to be toolbars, action bars, or drop-down lists of commands.

Creating a Menu

To define a menu from within MQL use the Add Menu command:

```
add menu NAME [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the menu.

You cannot have both a command and a menu with the same name.

`ADD_ITEM` is an Add Menu clause that provides additional information about the menu.

For more information, see *MQL Command Reference: menu Command* in the online user assistance.

Channels

Channels are essentially a collection of commands. They differ from menus in that they are not designed for use directly in an application, but are used to define the contents of a *portal*. Channels and portals are installed with the Framework and used in apps to display PowerView pages, but may also be created for use in custom Java applications.

The use of the term "Portal" here refers to the administration object, and not to the broader internet definition described in JSR 168.

Business Administrators can create channel objects if they have the portal administrative access. Since commands are child objects of channels, commands are created first and then added to channel definitions, similar to the association between types and attributes. Likewise, channels are created before portals and then added to portal objects. Changes made in any definition are instantly available to the applications that use it.

Channels created in MQL can optionally be defined as a user's workspace item.

Creating a Channel

To define a channel from within MQL use the add channel command:

```
add channel NAME [user USER_NAME] [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the channel.

You cannot have both a channel and a portal with the same name.

`USER_NAME` can be included if you are a business administrator with person/group/role access defining a channel for another user. This user is the item's "owner." If the user clause is not included, the channel is a system item.

`ADD_ITEM` is an add channel clause that provides additional information about the channel.

For more information, see *MQL Command Reference: channel Command* in the online user assistance.

Portals

A *portal* is a collection of *channel*s, as well as the information needed to place them on a Web page. Some portals are installed with the Framework and used in apps to display PowerView pages, but they may also be created for use in custom Java applications. Business Administrators can create portal objects if they have the portal administrative access.

The use of the term "Portal" in this chapter refers to the administration object, and not to the broader internet definition described in JSR 168. In headings in the documentation we use the terms 3DSpace portal and Public portal when a discernment needs to be made.

Before creating a portal, the channels that it will contain must be defined. Channels are child objects of portals — channels are created first and then added to portal definitions, similar to the association between types and attributes. Changes made in any definition are instantly available to the applications that use it.

Portals created in MQL can optionally be defined as a user's workspace item.

Creating a Portal

To define a portal from within MQL use the add portal command:

```
add portal NAME [user USER_NAME] [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the portal.

`USER_NAME` can be included if you are a business administrator with person/group/role access defining a channel for another user. This user is the item's "owner." If the user clause is not included, the portal is a system item.

`ADD_ITEM` is an add portal clause that provides additional information about the portal.

For more information, see *MQL Command Reference: portal Command* in the online user assistance.

Tables

Tables can be defined to display multiple business objects and related information. Each row of the table represents one business object. Expressions are used to define the columns of data that are presented about the business objects in each row. When you define a table, you determine the number and contents of your table columns.

There are two kinds of tables:

- *A User* table is a user-defined template of columns that can be used when objects are displayed in the Matrix Navigator (Details browser mode only), or in MQL with the Print Table command.

  User tables are created in MQL, or in Matrix Navigator's Visuals Manager and displayed when Matrix Navigator is in details mode. In Matrix Navigator, tables that users create are available from the Views/Tables menu. This enables viewing different information about the same object with a single mouse-click.

  Tables can be added as part of the definition for a customized View. Each time that View is activated, the Table defined for it displays.

  Users can save Table definitions by name (as personal settings) to turn on or off as needed. A single table may become part of several Views, being activated in one, and available in another. Refer to the *Matrix Navigator Guide: Using View Manager* for more information on Views.

  From Matrix Navigator browsers, tables that present information in a familiar format can be very useful. Each user can create her/his own tables from Matrix Navigator (or MQL). However, if your organization wants all users to use a basic set of similar tables consistently, it may be easier to create them in MQL, then copy the code to each user's personal settings (by setting context).
- *A System* table is an Administrator-defined template of columns that can be used in custom applications. These tables are available for system-wide use, and not associated with the session context. Each column has several parameters where you can define the contents of the column, link data (href and alt), user access, and other settings. For example, a user could click on a link called Parts to display a system table containing a list of parts. The other columns in the table could contain descriptions, lifecycle states, and owners.

  System tables are created by business administrators that have Table administrative access, and are displayed when called within a custom application.

  System table columns can be role-based, that is, only shown to particular users. For example, the Parts table might have a Disposition Codes column that is shown only when a person is logged in as a user defined as a Design Engineer. Or a user defined as a Buyer might be shown a column in a table that is not seen by a Supplier user. When no users are specified in the command and table definitions, they are globally available to all users.

When business objects are loaded into a table, Collaboration and Approvals evaluate the table expressions for each object, and fills in the table cells accordingly. Expressions may also apply to relationships, but these columns are only filled in when the table is used in a Navigator window. You can sort business objects by their column contents by clicking on the column header.

Creating a Table

To define a table from within MQL use the Add Table command:

```
add table NAME user USER_NAME [ADD_ITEM {ADD_ITEM}];
```

Or

```
add table NAME system [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to table.

`USER_NAME` can be included with the user keyword if you are a business administrator with person access defining a table for another user. I

`system` refers to a table that is available for system-wide use, and not associated with the session context.

You must include either the user or system keyword.

`ADD_ITEM` is an Add Table clause which provides additional information about the table.

For more information, see *MQL Command Reference: table Command* in the online user assistance.

Forms

A *form* is a window in which information related to an object is displayed. The Business Administrator designs the form, determining the information to be presented as well as the layout. Forms can be created for specific object types, since the data contained in different types can vary greatly. In addition, one object type may have several forms associated with it, each displaying different collections of information. When a user attempts to display information about an object by using a form, Collaboration and Approvals only offer those forms that are valid for the selected object.

Expressions can be used in the form's field definitions to navigate the selected object's connections and display information from the relationship (attributes) or from the objects found at their ends. Forms can be used as a means of inputting or editing the attributes of the selected object. However, attributes of related objects cannot be edited in a form.

A special type of form called a Web form can be created for use in custom applications. Each field has several parameters where you can define the contents of the field, link data, user access, and other settings.

You must be a Business Administrator to define a form, and have form administrative access.

Defining a Form

Use the Add Form command to define a form:

```
add form NAME [web] [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the form you are creating.

`web` is used when creating a "Web form." This distinguishes forms to be used in HTML/JSP applications from those used in the Studio Modeling Platform.

`ADD_ITEM` is an Add Form clause which provides additional information about the form.

For more information, see *MQL Command Reference: form Command* in the online user assistance.

7 Maintenance

This chapter discusses database maintenance issues.

Maintaining the System

The duties of the System Administrator are to maintain and troubleshoot the system. The System Administrator should also be the on-site point of contact for all software updates, revisions, and customer support requests. Specific duties include:

- *Controlling System-wide Settings*
- *Validating the 3DSpace Database*
- Maintaining and monitoring clients (See the *Administration Guide: Maintaining and Monitoring Clients*.)
- Monitoring server diagnostics (See the A*dministration Guide: Diagnostic Commands*.)
- *Developing a Backup Strategy*

Other responsibilities are covered in these chapters:

- *Studio Modeling Platform Overview: Building the System Basics* in the online user assistance
- *Working with Import and Export* in Chapter 8

Controlling System-wide Settings

Password requirements can be set for the entire system. For more information, see *Studio Modeling Platform Overview: Controlling Access: User Access: User Passwords* in the online user assistance. Triggers may be turned on or off with the triggers off command. In addition, System Administrators can control certain other settings that affect the system as a whole using the set system command:

```
set system SYSTEM_SETTING;
```

The specific settings are discussed below.

For more information, see *MQL Command Reference: set system Command* in the online user assistance.

Case-Sensitive Mode

Collaboration and Approvals by default is case-sensitive. However, when using Oracle Enterprise Edition, Oracle Standard Edition, or SQL Server, Collaboration and Approvals can be configured to operate in case-insensitive mode.

Oracle Standard Edition is not available for 64-bit operating systems. Oracle Enterprise Edition is recommended for maximum scalability and performance.

The schema must be at version 10.5 or higher to turn `casesensitive` off.

Oracle setup

New and existing Oracle databases must be set up to use a function-based index, if you want to work in a case insensitive environment.

To configure Oracle to work in a case-insensitive mode

1. Modify the Collaboration and Approvals user to add the QUERY REWRITE system privilege:
   ```
   grant query rewrite to Matrix;
   ```

2. In the init<SID>.ora file, add the following lines:
   ```
   QUERY_REWRITE_ENABLED=TRUE
   QUERY_REWRITE_INTEGRITY=TRUSTED
   ```
3. You must configure Collaboration and Approvals to turn off case-sensitivity, as described in *3DSpace setup*.

SQL Server Setup

By default, SQL Server is supported as case-insensitive. Case sensitivity is set by configuring the character collation set of the underlying database.

To configure SQL Server to work in case-sensitive mode

1. Execute the following SQL to get the available collations. Case-sensitive collations will have "CS" in the name.
   ```
   select * from fn_helpcollations()
   ```
   For example, Latin1_General_CS_AS is a case-sensitive Latin1 collation.
2. For a new database, use the selected collation to configure the database:
   ```
   create database db_name collate collation
   ```
   Where db_name is the name of your database and collation is the selected collation.

   For an existing database, use the selected collation to configure the database:
   ```
   alter database db_name collate collation
   ```
   Where db_name is the name of your database and collation is the selected collation.
3. Verify that case-sensitivity is turned On by executing the following commands in MQL. (You should do a `clear all` before executing these commands):
   ```
   set context user creator;
   set system casesensitive on;
   print system casesensitive;
   CaseSensitive=On
   ```
4. Close MQL and restart it.

It is very important that MQL be closed and restarted after setting the `system casesensitive` setting to On.

3DSpace setup

3DSpace/Oracle databases must be at schema Version 10.5 or higher to be able to turn case sensitivity off. For databases older than 10.5, you must run the MQL upgrade command with Version 10.5 or higher. You also must convert unique constraints to unique indices in Oracle.

There is the real possibility that duplicate records will exist if you have used the database before turning off the `casesensitive` setting. For example, in a case sensitive environment, you could have users named "bill" and "Bill", which would cause unique constraint violations if an attempt was made to make this database case insensitive (add unique indices). The MQL command `validate unique` is provided to identify these conflicts so that they can be resolved prior to turning off the setting. Once resolved, you then turn off the `casesensitive` system setting, which defaults to on, and reindex the vaults to switch to unique indices.

You can run the `validate unique` command against pre-10.5 databases using MQL 10.5 or higher to identify conflicts before upgrading.

To make a database case-insensitive

1. In MQL, run `validate unique` to identify conflicts and resolve any issues by changing some names or deleting unneeded items. For example, the output might show:

   ```
   Duplicate person 'Bill'
   Duplicate program 'test'
   Duplicate state 'Open' in policy 'Incident'
   Duplicate signature 'Accept Quality Engineer Assignment' on state
   'Assign' in policy 'Incident'
   Duplicate signature 'Accept Quality Engineer Assignment' on state
   'Assign' in policy 'Incident'
   Duplicate business type 'CLASS'
   Duplicate business object 'RequestReport' 'Unassigned' '1'
   Duplicate business object 'Document' 'Financials' 'Q32003'
   Duplicate business object 'Test Suite' 'Vault' ''
   Duplicate business object 'Milestone' 'Covers For Manuals' ''
   Duplicate business object 'Task' 'Regression testing' '0'
   ```

   You should find all of these objects (in Matrix Navigator, Business or System) and resolve them by deleting or changing the name of one of the duplicates. Note that duplicate signatures may be reported more than once. This is dependent on the number of unique signers on the signature for all actions (approve, reject, ignore).

   You may also find Person workspace objects such as:

   ```
   Duplicate table 'Cost PRS' owned by ganley
   Duplicate BusinessObjectQuery 'a' owned by coronella
   Duplicate VisualCue 'committed' owned by maynes
   Duplicate ObjectTip 'Description' owned by liu
   Duplicate Filter 'feature' owned by zique
   Duplicate BusinessObjectSet 'Current Software' owned by powers
   Duplicate ProgramSet 'New Bug' owned by oconnor
   ```

   These must be resolved by the user specified as the owner.

   All duplicates, including administrative objects, business objects, and workspace objects must be resolved before preceding.

2. Identify and resolve existing attribute and policy definitions to be sure the rules you set are what you want. See *Collaboration and Approvals with Casesensitive Off* for details.

3. In MQL, run `set system casesensitive off;`

4. Reindex all vaults to create case insensitive indices. For very large vaults this may take several hours. You can run `validate index vault VAULTNAME;` for information on what will occur. For more information, see *MQL Command Reference: vault Command: Index Vault* in the online user assistance.

If duplicates exist an error occurs during the vault index.

5. Calculate or update statistics on all tables by executing the following MQL command:

   ```
   <mql> validate level 4;
   ```

   The validate level 4 command should be run after any substantial data load operation, and periodically thereafter, in order to update the statistics. Refer to *Validating the 3DSpace Database* for details.

3DSpace is now ready to use in case-insensitive mode.

Adaplets

Adaplets generate SQL independently of 3DSpace, and they may be configured to be either case sensitive or insensitive. To make an adaplet operate in case insensitive mode, add the following line to the adaplet mapping file:

```
item casesensitive false
```

Without this line, adaplets will continue to behave in a case-sensitive fashion. Note that adding this line to an adaplet mapping file will cause 3DSpace to generate SQL against the foreign database that wraps all arguments inside of an 'upper()' expression. This alone does not assure the foreign database was designed and indexed in a way that makes case-insensitive operations viable.

The foreign database may need its own configuration to run in a case insensitive manner.

If the adaplet has case-sensitivity turned on and is in extend or migrate mode, then types must use mapped ids. Refer to the *Adaplet Programming Guide* for more information.

Collaboration and Approvals with Casesensitive Off

In general when user (or program) input that corresponds to a string in the database, a case insensitive search is performed (with case sensitivity turned off). If a match is found, the user input is replaced with the database string and the processing continues.

For example, selectables and the references within square brackets of select clauses are not case sensitive when in case insensitive mode. However, regardless of what you put inside the square brackets, in most cases the system returns the name as it is stored in the database. For instance, with a type called “test” that has an attribute “testattr”, the following command returns the output shown below.

```
print type test select attribute[TESTATTR};
  attribute[testattr]=TRUE;
```

Also, when case sensitivity is turned off, the query operators “match” and “matchcase” become indistinct, as are “!match” and “!matchcase.”

Some exceptions apply and are discussed below. Before turning case-sensitivity on, custom code should be checked to see if any routines check user input against database strings. These routines may need to be reworked to achieve the expected results.

Square Brackets

As stated above, in most cases the system returns the name of a selectable as it is stored in the database, regardless of what you put inside the square brackets. Actually, this applies only to references to administration objects that can be searched for in “Business” or “System”; when in square brackets these will return the database name of the object. However, things like file names, state names and signature names are returned as entered in the square brackets. For example, consider the following:

An object has an attribute Att1 and a checked in file name File.txt:

```
MQL<6>temp query bus * a2 * select format.file[file.txt]
format.file[file.txt].name attribute[att1].value;
businessobject A a2 0
    format.file[file.txt] = TRUE
```

(note that the spelling of the filename in the brackets is not corrected.)

```
format.file[file.txt].name = File.txt
```

(but, the name of the file is output as recorded in the database)

```
attribute[Att1].value = 2
```

(and the attribute name (an admin type) is corrected.)

One exception to this rule is properties on administration objects. Properties on administration objects always come back with the database name of the property.

Attribute Range Values

Before turning off case sensitivity, you should check all attributes for ranges that may disobey the rules of case insensitivity, such as having 1 range values for initial capitalization and another all lower case (for example “Pica” and “pica” for Units attribute). In this example, one of these ranges should be deleted. In a case-insensitive environment, the equals and match operators are identical; that is “Ea” is the same as “EA” and so entering the attribute value as “Ea” will succeed even if the range value is set to “EA”.

Revision Fields

When creating objects using a policy that defines an alphabetic revision sequence, the revision field is case sensitive, regardless of the system setting (that is, Collaboration and Approvals follows the revision rule exactly). Be sure the rules are defined appropriately.

However, queries on the revision field do follow the rules for the system `casesensitive` setting. So if your query specifies “A” for the revision field with case sensitivity turned off, objects that meet the other criteria are found that have both “A” and “a” as their revision identifier.

File names

File names remain case sensitive, regardless of the case sensitivity setting. So, for example, if the object has a file “FILE.txt” checked in, the following command will fail:

```
checkout bus testtype testbus1 0 file file.txt;
```

And the following will succeed:

```
checkout bus testtype testbus1 0 file FILE.txt;
```

For checkins, if you check in a file called “FILE.txt” and then check in another file called “file.txt”, there will be two separate files checked in (using append).

User Passwords

You cannot have distinct objects with the same spelling but different capitalization (that is, “Bill” and “bill” are interpreted as the same thing). However, user passwords remain case sensitive. For example, a user named “Bill” with a password of “secret”, can set context with:

```
mql<> set context user Bill pass secret;
```

or:

```
mql<> set context user bill pass secret;
```

but not with:

```
mql<> set context user bill pass Secret;
```

Adaplet usage

When an object name is explicitly specified in the name field of a query, objects returned via an adaplet are shown with the name in the case as entered in the query, and not necessarily as it exists in the database. The same is true when explicitly specifying object names to be exported — adaplet objects are referenced in the export file with the name in the case as entered.

Updating Sets With Change Vault

When an object’s vault is changed, by default the following occurs behind the scenes:

- The original business object is cloned in the new vault with all business object data except the related set data
- The original business object is deleted from the “old” vault.

When a business object is deleted, it also gets removed from any sets to which it belongs. This includes both user-defined sets and sets defined internally. IconMail messages and the objects they contain are organized as members of an internal set. So when the object’s vault is changed, it is not only removed from its sets, but it is also removed from all IconMail messages that include it. In many cases the messages alone, without the objects, are meaningless. To address this issue, the following functionality can optionally be added to the change vault command:

- Add the object clone from the new vault to all IconMail messages and user sets in which the original object was included.

Since this additional functionality may affect the performance of the change vault operation if the object belongs to many sets and/or IconMails, it is not part of the function by default, but business administrators can execute the following MQL command to enable/disable this functionality for system-wide use:

```
set system changevault update set | on  | off |;
```

For example, to turn the command on for all users, use:

```
set system changevault update set on;
```

Once this command has been run, when users change an object’s vault via any application (that is, Studio Modeling Platform or Web applications, or custom Studio Customization Toolkit applications), all IconMails that reference the object are fixed, as well as user’s sets.

Database Constraints

Some versions of Oracle have a bug that limits performance and concurrency when an operation uses a column that has a foreign key constraint and that column is also not indexed. Such columns can cause deadlocks and performance problems. For systems that use foreign keys extensively, this leads to a trade off that must be made between performance, storage, and use of foreign keys. You can use the set system constraint command to tailor the schema for one of three possible modes of operation::

```
set system constraint |none                      |;
                      |index [indexspace SPACE] |
                      |normal                    |
```

- Normal

  The normal system setting results in a schema that has foreign keys that are not indexed; that is, foreign key constraints are configured as they have been in pre-version 10 releases. This is the default setting. Databases using this setting are subject to the Oracle concurrency bug regarding non-indexed foreign keys.

- Index

  The index setting results in an Oracle index being added to every column that is defined as a foreign key, and has no existing index. Overhead in both storage and performance is added since updates to foreign keys also require updates to their corresponding index. This option should be used in development and test environments, where there is substantial benefit in the enforcement of foreign key constraints, and the negative storage/performance impact will not affect large numbers of users.

  When issuing the command to add indices to the system, you can specify the tablespace to use for the indexing operation.

- None

  This option improves concurrency by removing non-indexed foreign key constraints, and no additional Oracle index is required. This option eliminates the concurrency problem of foreign keys, and in fact, further improves system performance and scalability by eliminating the low-level integrity checks performed at the database level. This option should be used once an application has been thoroughly tested and is rolled out to a large-scale production environment.

When the system is set with a constraint mode, not only are the changes made to the relevant columns, but also the setting is stored in the database so that all future operations including creation of new tables, running the index vault command, and upgrade will use the selected mode.

Note that these options only affect approximately a dozen columns (among all tables) that have a foreign constraint but not an index.

Time Zones from the Database

You can use the GMT time setting in the database for your servers by running the following command:

```
set system dbservertimezonefromdb on;
```

When set, the system gets the time in GMT from the db server. GMT can also be determined based on the local time and time zone of the server object defined by the System administrator.

MX_DECIMAL_SYMBOL and System Decimal Symbol

Collaboration and Approvals has two distinct controls for the handling of period '.' or comma ',' as the decimal symbol for real numbers:

- The "set system decimal" setting determines how real values must be formatted to pass them to an Oracle database.
- The MX_DECIMAL_SYMBOL environment setting controls the format in which real numbers are returned as values in print statements, queries, expands, and selectables. This affects the strings returned by both MQL commands and Java ADK calls. The final display format can be further adjusted by application preferences and/or browser settings.
  This setting has no effect on the form of real numbers passed to database APIs, nor does it have to be the same as the "set system decimal" character.

The decimal separator mechanism works as follows:

- When accepting a real number as input to the database, the kernel replaces the decimal separator in real-number inputs with the "set system decimal" character before passing them along to the database server.
- When reading a real number from the database and returning it to the calling application, the kernel substitutes the decimal separator in real numbers with the MX_DECIMAL_SYMBOL character.

Because the "set system decimal" character controls the form in which you interact with the database server, this setting *must* be synchronized with the Oracle setting for NLS_LANG so that real numbers are passed in a form that is expected by the database interfaces.

Use the following MQL command to set the decimal character that is expected by the database server (according to the NLS_LANG setting):

```
set system decimal CHARACTER;
```

where CHARACTER can be . or , [period or comma].

For example, to set the system decimal to a comma, use:

```
set system decimal ,;
```

When setting the system decimal character, be sure to use the setting that is implied by the database's NLS_LANG setting. The default setting is period '.'. So, if the database setting for NLS_LANG is AMERICAN_AMERICA.WE8ISO8859P15, Oracle expects a period for the decimal symbol (as indicated by the territory setting of AMERICA), and 3DSpace makes the necessary conversion, once the following command is run:

```
set system decimal .;
```

When exporting business objects, MX_DECIMAL_SYMBOL influences the format of real numbers written to the export file. Therefore, the user who imports the file should set MX_DECIMAL_SYMBOL in the same way, or errors will occur.

The command `set system decimal CHARACTER;` *is not supported for use with databases other than Oracle. For non-Oracle databases, the data is always stored with '.' but displayed based on the desktop client's MX_DECIMAL_SYMBOL setting, or in Web based implementations the locale of the browser.*

For more information on configuring 3DSpace for multiple language support, see the following topics in the *Administration Guide*:

- *Localizing 3DSpace*
- *Language Support*
- *Configuring a Japanese Web Environment*

Allowing Empty Strings for Object Names

The following system-wide setting is available so that System Administrators can enable/disable the creation of objects with empty name fields in MQL for all user sessions permanently:

```
set system emptyname [on|off]
```

By default, the setting is off, which means that empty names are not allowed. MQL and any other program code will issue an error if any attempt to make a business object have an empty name is made. It will also cause a warning to be issued if a query specifies an empty string (“ ”) for the name. (The query will not error out and will not abort any transaction going on.) If the setting is on, the system allows empty names for objects in MQL only.

Setting History Logging for the System

The following system-wide setting is available to enable/disable history for all user sessions permanently:

```
set system history [on|off]
```

By default, history is on. When turned off, custom history records can be used on the operations where history is required, since it will be logged regardless of the global history setting.

Adaplet Persistent IDs

Adaplet object IDs are saved in the database by default so that they are persistent between sessions. This capability is no longer dependent on the usage mode (readonly, readwrite, migrate, extend) of the adaplet. To disable the feature, persistence may be turned off with the following command:

```
set system persistentforeignids off;;
```

Refer to the *Adaplet Programming Guide* for details.

Privileged Business Administrators

By default, a business administrator can change context to any person without a password. This is to allow administrators and programs to perform operations on behalf of another user. A System Administrator can disable this system-wide “back door” security risk by issuing the following command:

```
set system privilegedbusinessadmin off;
```

After this command has been run, business administrators need a password when changing context to another person. Only system administrators can change context to any other person without a password.

A System Administrator can re-enable business administrators to change context without a password using:

```
set system privilegedbusinessadmin on;
```

System Tidy

In replicated environments, when a user deletes a file checked into an object (via checkin overwrite or file delete), by default all other locations within that store maintain their copies of the now obsolete file. The file is deleted only when the administrator runs the `tidy store` command.

You can change the system behavior going forward so that all future file deletions occur at all locations by using:

```
set system tidy on;
```

Since this command changes future behavior and does not cleanup existing stores, you should then sync all stores, so all files are updated. Once done, the system will remain updated and tidy, until and unless system tidy is turned off.

Running with system tidy turned on may impact the performance of the delete file operation, depending on the number of locations and network performance at the time of the file deletion.

Validating the 3DSpace Database

The MQL `validate` command enables you to check the correctness and integrity of the 3DSpace database. See the S*erver Installation Guide: Upgrading the Database after Installing a New Version of Software* for complete syntax and clauses for working with the MQL validate command.

Correct command

The MQL correct command can be used by System Administrators as directed by Dassault Systèmes Technical Support, to correct anomalies caused by older versions of the product. You must be certain that no users are logged in or can log in while correct is running.

Confirm that no users are connected, using the sessions command or Oracle tools, and temporarily change the bootstrap so that no one can login while correct is running.

There are several forms of the command. Each form is described in the sections that follow.

Correct Vault

The correct vault command can be used to validate or correct the following relationship issues in a database:

1. “Stale” relationships - Relationships that reference a business object that does not exist.

   Since there is no way to determine which business object should be referenced, the relationship is deleted when the correct command is issued with “fix”.

   A sample of the output generated when fixing this issue is shown below:

```
// Missing to/from businessobjects
From vault Zephyr to vault Zephyr
From vault Zephyr to vault KC
remove stale relationship
  type: BrokenRel
  from: 8286.42939.11584.38392
  to: BrokenRel child1 0
remove stale relationship
  type: BrokenRel
  from: 8286.42939.11584.38392
  to: BrokenRel child1 0
From vault KC to vault Zephyr
From vault KC to vault KC
Correct finished in 0.12 second(s)
Total of 2 problem(s) encountered and fixed
```

   If in validate mode, the last line would say:

```
Total of 2 problem(s) encountered and validated
```

2. extra “to” end - A cross vault relationship has a valid “to” side object, but there is no corresponding “from” object.

   The relationship is deleted when the correct command is issued with “fix” as there is insufficient information to reconstruct the relationship - its attributes and history are missing.

A sample of the output generated when fixing this issue is shown below:

```
// Missing entry in the lxRO table of the “from” end of the rel
From vault Zephyr to vault Zephyr
From vault Zephyr to vault KC
From vault KC to vault Zephyr
remove extra relationship 'to' end
     type: BrokenRel
     from: BrokenRel child1 0
     to:   BrokenRel TO 0
From vault KC to vault KC
Correct finished in 0.12 second(s)
Total of 1 problem(s) encountered and fixed
```

3. missing “to” end - A cross vault relationship has a valid “from” side object, but there is no corresponding “to” object.

   Missing “to” end’s are recreated when the database is corrected. This is possible because only the relationship pointers need to be re-established. The attribute and history for the relationship is stored on the “from” side.

   A sample of the output generated for this case is shown below:

```
// Case 3
// Missing entry in the lxRO table of the 'to' end of the rel
From vault Zephyr to vault Zephyr
From vault Zephyr to vault KC
add missing relationship 'to' end
   type: BrokenRel
   from: BrokenRel FROM 0
    to:   BrokenRel child1 0
From vault KC to vault Zephyr
From vault KC to vault KC
Correct finished in 0.11 second(s)
Total of 1 problem(s) encountered and fixed
```

Cases (1) and (2), by their nature, may result in data being removed from the database. Case (3) is an additive step; such cases do not result in the loss of information.

It should be noted that you can validate and correct adaplet vaults used in extend or migrate mode; read or readwrite adaplet vaults are not supported.

Syntax

The syntax of the correct vault command is:

```
correct vault VAULTNAME [to VAULTNAME] [type REL1{,REL2}] [verbose]
[fix] [validate];
```

Unless “fix” is specified, the command will run in “validate” mode.

If you include both “fix” and “validate,” an error will occur asking you to specify just one of the options. If you use “correct validate” without any arguments, all relationships’ from objects are checked and a report is generated on stdout that lists all problems found.

Each argument is described below.

vault VAULTNAME [to VAULT]

Limits the scope of the function to a single vault. All relationships that include any object in VAULTNAME are checked. You can also specify “all” for correct to go through all the vaults in the database.

When “to VAULT” is included, correct looks only at connections between VAULTNAME and VAULT.

type REL1{,REL2}

Inspects only relationship types listed. If not specified, all the relationship types in the system will inspected for problems.

verbose

Causes the program to output the SQL needed for repair to stdout. Set to false by default.

fix

Fixes the database. Set to false by default.

validate

Validates the database. Reports broken relationships but no corrections are made. Set to true by default.

Transactions

The correct vault command operates in two passes. The first pass is a quick scan through the database without performing any row level locks. The second pass gathers detailed information on relationships reported as “suspect” during the first pass.

During concurrent use of the system, it is entirely possible that the validate function on a vault may report errors that do not actually exist in the database since it is operating without the benefit of locks (the data many be changing while it is being examined). However, during the second pass, any suspicious data is fully re-qualified using row level locks, making the function safe to use while the system is live.

Note that with any general purpose application that uses locks, a deadlock situation may occur if other applications are locking the same records in an inconsistent fashion. Deadlocks are highly unlikely while using correct vault since the data that is being locked (e.g. a 'stale' relationship) is generally unusable by other applications. However, if deadlock should occur, simply reissue the correct vault command and allow it to run to completion.

This two pass transaction model is only used by the correct vault command. While "correct vault" is designed to run while in concurrent use, it is recommended that for all system administrator operations like this one, it be issued when there are no other users logged on.

Whenever any form of the correct command is executed in fix mode, you receive the following warning and prompt for confirmation:

```
Correct commands may perform very low level modifications to your
database.
Use the 'sessions' command to check that no users are logged in.
It is strongly recommended that you change bootstrap password to insure
that no users log in during correct process, and turn verbose on to
record corrective actions.
```

```
You should also consult with MatrixOne support to insure you follow
appropriate procedures for using this command.

Proceed with correct (Y/N)?y
```

For all correct commands, output can be redirected to a file using standard output.

Correct Attribute

To verify the integrity of attributes and types, each type in the database can first be checked to ensure that no attribute data or attribute range values are missing. The following command will return a list of objects of the specified TYPE that have missing data:

```
MQL<> correct attribute validate type TYPE;
```

where TYPE is a defined type in 3DSpace. If corrections must be made, MQL will provide the business object type, name, and revision, as well as what attribute data is missing, as follows:

```
Business object TYPE NAME REVISION is missing attribute ATTRIBUTE_NAME
```

Be sure to run this command for each type in the database.

If any business objects are returned by the `correct attribute validate` command, those business objects must be fixed with the following command:

```
MQL< > correct attribute modify type TYPE_NAME attribute
ATTRIBUTE_NAME vault VAULT_NAME;
```

Where TYPE_NAME is the name of a business type to be corrected. When this command is completed all business objects that have missing attributes will have those attributes back. The value of the attribute will be the default value.

Where ATTRIBUTE_NAME is the name of the attribute that is missing from some objects.

Where VAULT_NAME is name of the vault in which business objects of type TYPE_NAME will be examined and corrected if needed.

Run this command for each vault in the database. Only those business objects that were missing the attribute will be modified. The added attribute on these objects will be initialized to its default value, and MQL will output confirmation messages similar to:

```
count of BOs of type 'TYPE'  in vault 'VAULT_NAME' = 1
   processing attribute = 'ATTRIBUTE_NAME'
        business obj '51585.1271.52275.16291' having db id '-869056605' with type
'TYPE_NAME'  is missing attr 'ATTRIBUTE_NAME'
           Total objects processed = 1
```

Correct Relationship

Correct relationship is used to correct relationship attribute data:

```
mql<> correct relationship;
```

Attributes on relationships that connect objects in different vaults are stored in the from-side vault. This command removes attributes if they exist in the to-side vault.

Correct Set

Sets can be validated either across the entire database, or on a server by server basis using the following:

```
MQL<> correct set validate [server SERVER_NAME];
```

If invalid sets are found you can delete them with:

```
MQL<> correct set fix [server SERVER_NAME];
```

Correct State

To verify the integrity of all states in the database or on a single vault use:

```
MQL<> correct state validate [vault VAULT_NAME];
```

Without the vault clause all states in the database are checked. If you include the vault clause, only that vault is validated. You can then fix the states using:

```
MQL<> correct state [vault VAULT_NAME];
```

Developing a Backup Strategy

Because there are a number of factors that can cause a database failure, including power loss, hardware failure, natural disasters, and human error, it is important that you develop both a backup and a recovery plan to protect your database. It is not enough that you *develop* a recovery plan, however. You must also test that recovery plan to ensure that it is adequate before your data is compromised. Finding out that your recovery plan is inadequate after you have already lost your data will not do you much good. Also, testing of the recovery plan may indicate changes you need to make to your backup strategy.

Inventories of all stores should be performed nightly as part of the backup procedure. For more information, see MQL Command Reference: store Command: Inventory Store Command in the online user assistance.

Working with Indices

3DSpace stores each attribute and “basic” property of an object in a separate row in the database, allowing flexible and dynamic business modeling capabilities. (Basic properties include type, name, revision, description, owner, locker, policy, creation date, modification date, and current state). This has the side effect of requiring extra SQL calls (joins) when performing complex queries and expands that specify multiple attribute or basic values. To improve performance of these operations, you can define an *index*. In fact, test cases show marked improvement on many types of queries and expands when an index is in place.

Queries and expands will choose the “best” index to perform the operation. If 2 indices are equally qualified to perform the operation, the first one found will be used.

An *index* is a collection of attributes and/or basics, that when enabled, causes a new table to be created for each vault in which the items in the index are represented by the columns. If a commonly executed query uses multiple attributes and/or basics as search criteria, you should consider defining an index. Once enabled, searches involving the indexed items generate SQL that references the index table instead of the old tables, eliminating the need for a join.

A query that doesn’t use the first specified item in an index will not use that index. Therefore, when creating an index, specify the most commonly used item first.

When an index is created or items are added or removed from it, the index is by default disabled. The new database tables are created and populated when the index is enabled. This step can be time-consuming; however, it is assumed that an index will be enabled by a system administrator once when created or modified, and will remain enabled for normal system operation.

For expands, 3DSpace looks for and uses an index that is associated with relationships; that is, an index that has a relationship attribute as the first item in it.

Creating an index is only one way of optimizing performance. A properly tuned database is critical. While the *Installation* and *MQL Guides* provide some guidelines and procedures, refer to your database documentation for tuning details.

Write operations of indexed items occur in 2 tables instead of just 1 and so a performance penalty is paid. This has a negligible effect in cases where end users may be modifying a few values at a time, but you should disable indices prior to performing bulk load/update operations. Re-enabling the indices after the bulk update is more efficient than performing the double-bookkeeping on a value-by-value basis during the bulk update.

Considerations

Before creating indices consider how the indexed items are used in your implementation. Some pros and cons are shown below

PROS	CONS
Excellent for reads.	Write performance is impacted (data is duplicated).
Query execution runtimes are significantly reduced	More disc space is required (for duplicate data).
Table joins are significantly reduced.	Greater potential for deadlock during write operations. (more data in one row as opposed to separate rows within separate tables).
	Item order in the definition of the index is important, since if the operation doesn't use the first item, the index is skipped.

Also, keep in mind that attributes that have associated rules cannot be included in an index.

Defining an Index

An index is created with the add index command:

```
add index NAME [unique] [ADD_ITEM {ADD_ITEM}];
```

`NAME` is the name you assign to the index.

`ADD_ITEM` is an add index clause which provides additional information about the index.

For more information, see MQL Command Reference: index Command in the online user assistance.

Enabling an Index

An index must be enabled before it is used to improve performance. The process of enabling an index generally takes 0.01 second per object in the database. Some rough processing times that can be expected are shown in the table below:

Number of objects in DB	Time to enable an Index
500,000	1.5 hours
5,000,000	14 hours

To optimize the performance, validating the tables is recommended. You should disable an index before performing bulk loads or bulk updates.

Creating and enabling indices are not appropriate actions to be performed within explicit transaction boundaries, particularly if additional operations are also attempted before a commit.

Validating an Index

Since up-to-date database statistics are vital for optimal query performance, after enabling an index you should generate and add statistics to the new database tables. This is assuming statistics are already up-to-date for all other tables. Refer to *Validating the 3DSpace Database* for more information.

Using the index as Select Output

You can use the index[] selectable on business objects and relationships to retrieve the attribute and basic values directly from the index table. This is roughly N times faster than using attribute[] where N is the number of attributes in the index. For example, if you routinely run a query/select like the following:

```
temp query bus Part * * select attribute\[Part_Number\]
attribute\[Quantity\] attribute\[AsRequired\]
attribute\[Process_Code\] attribute\[Function_Code\] current dump |
output d:/partlist.txt;
```

You could create an index called Parts containing all of the selectables listed and use the following query instead:

```
temp query bus Part * * select index\[Parts\] dump | output d:/
partlist.txt;
```

The temp query that uses the select index is roughly 6 times faster than using the query that selects all 6 items separately.

Select index is similar to selecting items with a few differences:

- Select index returns values for all items in the index, where select item returns just one. If an attribute does not apply to a type or relationship, the returns is null.
- String attributes longer than 251 characters will be truncated to 251 characters. Generally these are defined as multiline attributes, which cannot be included in an index anyway.
- Descriptions longer than 2040 characters will be truncated to 2040 characters.

Index Tracing

Tracing can be enabled to detect queries/expands that can benefit by using a new or modified index. When enabled, messages are displayed whenever a query or expand is executed that fails to find an associated index. A message is output for each item in the query or expand that *could* be in an index (an attribute or basic property). The intent of this tracing is to highlight queries and expands that use indexable items, none of which are the first item in any index (so no index is used to optimize the operation).

To enable the index trace messages, use the MQL command:

```
trace type INDEX on;
```

or set the environment variable:

```
MX_INDEX_TRACE=true
```

When this trace type is enabled, trace messages will be displayed in the format:

```
No index found for attribute[NAME1],attribute[NAME2],BASIC3…
```

The items in the trace message may or may not be in a defined index. However, even if they are part of an enabled index, they are not the first item in it, and so the index is not usable for that query or expand.

A developer can conclude that adding at least some of the listed fields to a new or existing index (making one of the items first) may improve system performance.

8 Working with Import and Export

This chapter discusses concepts about exporting and importing administrative and business objects.

In this section:

Overview of Export and Import

Administrative definitions, business object metadata and workflows, as well as checked-in files, can be exported from one root database and imported into another. Exporting and importing can be used across schema of the same version level, allowing definitions and structures to be created and “fine tuned” on a test system before integrating them into a production database.

When you export a business object, you can save the data to a *exchange* file or to an XML format file that follows the Matrix.dtd specification. An exchange file is created according to the exchange Format. This format or the XML format must be adhered to strictly in order to be able to import the file.

The Exchange Format is subject to change with each version.

Exporting

Before exporting any objects, it is important to verify that MQL is in native language mode. Exporting in the context of a non-native language is not supported.

Export Command

Use the Export command to export the definitions of a database. The complete export command for administrative objects is as follows:

```
export ADMIN_TYPE TYPE_PATTERN [OPTION_ITEM [OPTION_ITEM]...]| into | file FILE_NAME
                                                             | onto |
[exclude EXCLUDE_FILE] [log LOG_FILE] [exception EX_FILE];
```

For more information, see *MQL Command Reference: export Command* in the online user assistance.

XML Export Requirements and Options

Schema and business objects may optionally be exported in XML format. The resulting XML stream follows rules defined in the XML Document Type Definition (DTD) file (named “ematrixml.dtd”) that is installed with 3DSpace in the /XML directory. This DTD file is referenced by all exported 3DSpace XML files. In order to interpret and validate 3DSpace XML export files, an XML parser must be able to access the eMatriXML DTD file. This means that the DTD file should reside in the same directory as the exported XML files and be transferred along with those files to any other user or application that needs to read them.

Using MQL, you can export objects to XML format in either of two ways:

- Insert an XML clause in any Export command for exporting administrative or business objects
- Issue an XML command to turn on XML mode as a global session setting. In this mode, any Export commands you enter, with or without the XML clause, will automatically export data in XML format.

Be aware of the following restrictions related to exporting and importing in XML format:

- When exporting programs to XML, make sure the code does not contain the characters “]]>”. These characters can be included as long as there is a space between the bracket and the greater sign.
- Legal characters in XML are the tab, carriage return, line feed, and the legal graphic characters of Unicode, that is, #x9, #xA, #xD, and #x20 and above (HEX). Therefore, other characters, such as those created with the ESC key, should not be used for ANY field, including business and administrative object names, description fields, program object code, or page object content.
- Due to an xerces system limitation, the form feed character “^L” is not supported by the import command. If they are included in the file you are trying to import, you will receive an error similar to:

  ```
  System Error: #1500127: 'file' does not exist
  businessobject Document SLB EH713116 AE failed to be imported.
  System Error: #1600067: XML fatal error at (file '/rmi_logs/GP2_tmp/
  T5011500_4x.xml', line 6292824, char 2): Invalid character (Unicode:
  0xC)
  ```

`import failed.`

The workaround is to go to the line number in the XML file and remove the offending character.

The CDATA termination character string]]> is replaced by]Inserted_by_ENOVIA]Inserted_by_ENOVIA> in XML export files. XML Import files will do back replacement. Outside software reading, XML files should handle this replaced string.

XML Clause

Use the XML clause as an OPTION_ITEM in any Export command to export administrative or business objects in XML format. To ensure that the XML file(s) reside in the same directory as the XML DTD, you can export files to the XML folder (in your ENOVIA_INSTALL directory) where the DTD is installed. Alternatively, you can specify another location in your Export command and copy the DTD into that directory. Use the `into` clause in your Export command and specify a full directory path, including file name. For example:

```
export person guy xml into file
c:\enoviaV6R2011\studio\xml\person.xml;
```

XML Command

Use the XML command to turn XML mode on or off as a global session setting. The complete XML command syntax is:

```
xml [on|off];
```

Omitting the on/off switch causes XML mode to be toggled on or off, depending on its current state. XML mode is off by default when you start an MQL session.

For example, to export a person named “guy” using an XML command along with an Export command, you can enter:

```
xml on;
export person guy into file
c:\enoviaV6R2011\studio\xml\person.xml;
```

If you need to export several objects to XML format, using the XML command to turn on XML mode first can eliminate the need to re-enter the XML clause in each Export command as well as the possibility of an error if you forget.

XML Output

The XML export format typically generates a file 2 to 3 times larger than the standard export format. To conserve space, subelements are indented only if verbose mode is turned on. Indentation makes output more readable but is not required by an XML parser. Some XML viewers (like Internet Explorer 5.0) will generate the indentation as the file is parsed.

The following example shows standard export output when you export a person named “guy” during an MQL session. While relatively compact, this output is not very intelligible to a user.

```
MQL<1>set context user creator;
MQL<2>export person guy;
!MTRX!AD! person guy 8.0.2.0
guy Guy ""
 "" "" "" "" 0 0
 1 1 1 0 1 0 1 "" "" *
 1111101111111100111110
 0 1 .finder * GuyzViewTest 0 * FileInDefaultFormat 1 ""
 0 0
 0 0
 0
 0
 0
 de3IJEE/JIJJ.
 ""
 0
 0
 0
 1
 Test""
1
Description  description 1
0 0 0 0 70 21 0 0 1 1
 1
 0 0
 0
 0 0
person guy successfully exported.
!MTRX!END
export successfully completed
```

The next example shows XML output when you use the Export command, with XML mode turned on, to export a person named “guy” during a continuation of the same MQL session. While more intelligible to a user, this code creates a larger output file than the standard export format.

```
MQL<3>xml on;
MQL<4>verbose on;
MQL<5>export person guy;
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- (c)MatrixOne, Inc., 2000 -->
<!DOCTYPE ematrix SYSTEM "ematrixml.dtd">
<ematrix>
  <creationProperties>
    <release>8.0.2.0</release>
    <datetime>2000-05-05T17:57:19Z</datetime>
    <event>export</event>
    <dtdInfo>&ematrixProductDtd;</dtdInfo>
  </creationProperties>
  <person id="0.1.31721.46453">
    <adminProperties>
      <name>guy</name>
    </adminProperties>
    <fullName>Guy</fullName>
    <fullUser/>
    <businessAdministrator/>
    <systemAdministrator/>
```

```
<applicationsOnly/>
<passwordChangeRequired/>
<access>
  <all/>
</access>
<adminAccess>
  <attributeDefAccess/>
  <typeAccess/>
  <relationshipDefAccess/>
  <formatAccess/>
  <personAccess/>
  <roleAccess/>
  <associationAccess/>
  <policyAccess/>
  <programAccess/>
  <wizardAccess/>
  <formAccess/>
  <ruleAccess/>
  <siteAccess/>
  <storeAccess/>
  <vaultAccess/>
  <serverAccess/>
  <locationAccess/>
</adminAccess>
<queryList count="1">
  <query>
    <name>.finder</name>
    <queryStatement>
      <vaultPattern>*</vaultPattern>
      <typePattern>GuyzViewTest</typePattern>
      <ownerPattern>*</ownerPattern>
      <namePattern>FileInDefaultFormat</namePattern>
      <revisionPattern>1</revisionPattern>
    </queryStatement>
  </query>
</queryList>
<password>de3IJEE/JIJJ.</password>
<tableList count="1">
  <table>
    <name>Test</name>
    <columnList count="1">
      <column>
        <label>Description</label>
        <expression>description</expression>
        <usesBusinessObject/>
        <geometry>
          <xLocation>0.0</xLocation>
          <yLocation>0.0</yLocation>
          <width>70.0</width>
          <height>21.0</height>
          <minWidth>0.0</minWidth>
          <minHeight>0.0</minHeight>
          <autoWidth/>
          <autoHeight/>
        </geometry>
        <editable/>
      </column>
    </columnList>
  </table>
```

```
    </tableList>
  </person>
</ematrix>
```

Importing

When migrating objects or entire databases, it is important to import in the correct order:

1. Import all administrative objects first.
2. Import all business objects.
3. Import workspaces from the same exported ASCII data file as was used to import Persons. Refer to *Importing Workspaces* for more information.
4. Import properties of the administrative objects that have them. The same file that was used to import the administrative objects should be used. Refer to *Importing Properties* for more information.

For more migration strategies refer to *Migrating Databases*.

Import Command

Use the Import command to import administrative objects from an export file to a database. The export file must be in Exchange Format or in an XML format that follows the Matrix.dtd specification.

```
import [list] [property] ADMIN_TYPE TYPE_PATTERN [OPTION_ITEM [OPTION_ITEM]...]
from file FILENAME [use [FILE_TYPE FILE [FILE_TYPE FILE]...];
```

For more information, see M*QL Command Reference: import Command* in the online user assistance.

Importing Servers

If a server object is imported (via MQL or Oracle) into a different Oracle instance or user than it was exported from, the username and password settings of the server must be modified before distributed access is available. This is particularly important if an upgrade will follow the import, since all servers must be accessible to the machine doing the upgrade.

Importing Workspaces

Workspaces contain a user's queries, sets, and IconMail, as well as all Visuals. Workspaces are always exported with the person they are associated with. However, when Persons are imported, the workspace objects are not included. This is because they may rely on the existence of other objects, such as Types and business objects, which may not yet exist in the new database. Workspaces must be imported from the same .mix file that was used to import persons. For example:

```
import workspace * from file admin.mix;
```

Or:

```
import workspace julie from file person.mix;
```

Importing Properties

Properties are sometimes created to link administrative objects to one another. Like workspaces, properties are always exported with the administrative object they are on. Import is somewhat different, however, since a property may have a reference to another administrative object, and there is no way to ensure that the referenced object exists in the new database (administrative objects are sometimes exported and then imported in pieces). So a command to import administrative objects is issued, the specified objects are created first, and then the system attempts to import its properties. If the "continue" modifier is used, the system will get all the data it can, including system and user properties. But to ensure that all properties are imported, even when the administrative data may have been contained in several files, use the `import property` command.

For example, data can be exported from one database as follows:

```
export attrib * into file attrib.exp;
export program * into file program.exp;
export type * into file type.exp;
export person * into file person.exp;
```

Then the administrative objects are imported:

```
import attrib * from file attrib.exp;
import program * from file program.exp;
import type * from file type.exp;
import person * from file person.exp;
```

Finally, workspaces and any "missed" properties are imported. Note that properties may exist on workspace objects, so it is best to import properties after workspaces:

```
import workspace * from file person.exp;
import property attrib * from file attrib.exp;
import property program * from file program.exp;
import property type * from file type.exp;
import property person * from file person.exp;
```

Importing Index objects

When importing an Index that includes attributes, these attributes must already exist in the new database — that is, you should import the attributes first. For example:

```
import attribute A from file /temp/export.exp;
import attribute B from file /temp/export.exp;
import index * from file /temp/export.exp;
```

If the export file contains a lot of other admin data and you want to "import admin *", you can use import options to avoid errors when the attributes are processed. For example, if you know you want the objects in the export file to supersede any objects of the same type/name in the database use the following:

```
import admin * overwrite from file /temp/export.exp;
```

If you want objects already in the database to remain unchanged, use:

```
import admin * commit 1 continue from file /temp/export.exp;
```

Extracting from Export Files

Sometimes export files contain more information than you want imported. When this is the case, the extract command can be used to create a new file containing only the specified information of the original file.

```
extract |bus OBJECTID     | [OPTION_ITEM [OPTION_ITEM]]from file FILENAME |into| file NEW;
        |ADMIN ADMIN_NAME|                                                 |onto|
```

Migrating Databases

When migrating entire databases or a large number of objects, it is important to import in the following order:

1. Import all administrative objects first. For example:

```
import admin * from file admin.mix;
```

2. Import all business objects. For example:

```
import bus * * * from file bus.mix;
```

3. Import workspaces from the same exported ASCII data file as was used to import Persons. For example:

```
import workspace * from file admin.mix;
```

Or:

```
import workspace julie from file person.mix;
```

Workspaces contain a user's queries, sets, TaskMail, and IconMail, as well as all Visuals. Workspaces are always exported with the person with which they are associated. However, when Persons are imported, the workspace objects are not included. This is because they may rely on the existence of other objects, such as Types and business objects, which may not yet exist in the new database.

4. Import properties of administrative objects.

   When administrative objects are imported, 3DSpace imports all the objects without their properties and then goes back and imports both system and user properties for those objects. If administrative objects were exported in pieces, such as all Types in one file, all Persons in another, then properties should be explicitly imported with the `import property` command. Refer to *Importing Properties* for more information.

Migrating Files

When migrating business objects, there are three options for its files:

1. By default, both file metadata and actual file contents are written to the export file for business objects exported with their revision chain.
   3DSpace UUencodes the file and writes it to the export data file, along with business object metadata. Pointers to the files are guaranteed because the file is recreated during import. In this case, any file sharing is lost, (as when revisions use the same file list—each revision in the chain gets its own copy of the file).
2. Add the `!file` clause to not include any file metadata or content.
3. Use the `!captured` clause to not include captured store file content.
   This option writes only the fully qualified path of checked in files in the data file, along with business object metadata.

When migrating databases, most often the `!captured` option is recommended. This will facilitate the process, in that the .mix files will not be as large, and therefore will not require as much disk space or processing time to complete the migration. Once the import is complete, the objects will point to the appropriate files in the same location.

The key to keeping the file pointers accurate is keeping the store path definitions consistent. For example, let's say the database from which we are exporting has a captured store named "Released Data Store." The path of this store is defined as "/company/released." To maintain pointer consistency when using `!captured`, the new database must also have a defined captured store "Released Data Store" with the same path definition.

If objects are to be deleted and then re-imported, use the `!captured` option, but be sure to tar off any captured store directories before deleting any objects. Once the objects are deleted, the directories restored, and the objects imported, the files will be associated with the appropriate objects.

Migrating Revision Chains

Files may be shared among revisions of a business object. This "file sharing" concept was introduced to minimize storage requirements and is primarily used with captured stores. However, this does mean special attention is required when exporting and importing. Consider the following:

If the entire revision chain is not going to be exported and/or imported into a new database, then the use of `!captured` may result in lost files. For example, if the business object Assembly 123 0 has three files checked in, and is then revised to Assembly 123 1, this new revision shares the three files with the original. As long as both are exported and imported, `!captured` can be used (in fact, should be used to avoid file duplicating). However, if just Assembly 123 1 is imported into a new database, then NO file is imported, because Assembly 123 1 inherits files from the previous revision, which is not in the database yet. If you import Assembly 123 0 later, Assembly 123 1 shows files.

Also, if you want to import business objects that have revisions and put them into a different vault, you *must* use a map file. If you attempt to use the `from vault` clause with the `to vault` clause, errors will occur.

Comparing Schema

This section describes how to compare two schemas to determine the differences between them. A *schema* is all the administrative objects needed to support an app. Use schema comparison to compare:

- Different versions of the same schema to manage changes to the schema.
- Two schemas from different databases so you can merge the schemas (for example, merge a checkout system with a production system).

The process of comparing schema involves two main steps:

1. Create a baseline sample of one schema using XML export. See *Creating a Baseline*.
2. Analyze the differences between the baseline sample and the other schema (or a later version of the same schema) using the compare command. You specify the administrative types (attributes, relationships, etc.) and a name pattern (for example, all attributes beginning with "Supplier") to compare. Each compare command outputs a single log file that contains a report. The report lists the administrative objects in the baseline export file that have been added, deleted, and modified. See *Comparing a Schema with the Baseline*.

If your goal is to merge the two schemas by making the necessary changes to one of the schemas (sometimes called "applying a delta"), you can make the changes manually or by writing an MQL script file that applies the changes to the target schema.

Scenario

Suppose you need to determine the changes that have occurred in a checkout database versus what continues to exist in the production database. In this case, you may want to create a baseline of both databases, and use each to compare against the other. One report would be useful to find out what has changed in the checkout database. The other report would be useful to determine what it would take to apply those changes to the production database.

Creating a Baseline

The first step for comparing two schemas is to establish a baseline for analysis by sampling one of the schemas. You create a baseline by exporting administrative objects to XML format. You can use any option available for the export command to create the baseline export files. For information on options and more details about the export command, see *Export Command*.

Use the following guidelines to perform the export.

- Start MQL using a bootstrap file that points to the database containing the schema for which you want to create the baseline.
- There are two ways to produce an XML export file: toggle on XML mode and issue a normal export command, or issue a normal export command but include the XML keyword. For example:

  ```
  xml;
  export ADMIN_TYPE TYPE_PATTERN into file FILENAME;
  ```

  Or the equivalent:

  ```
  export ADMIN_TYPE TYPE_PATTERN xml into file FILENAME;
  ```
- It's best to create separate a export file for each administrative type and to keep all the objects of a type in one file. For example, export all attributes to file attributes.xml, all relationships to relationship.xml, etc. This keeps the baseline files to a reasonable size, and also lets you compare specific administration types, which makes it easy to produce separate reports for each administration type. If you need to identify subsets of objects within an export file to focus the analysis, you can do so using the compare command.
- The compare command requires that the ematrixml.dtd file be in the same directory as the export files. Therefore, you should create the export files in the directory that contains the dtd file or copy the dtd file into the directory that contains the export files. If you don't specify a path for the export file, 3DSpace creates the file in the directory that contains mql.exe file.

The following table shows examples of export commands that export different sets of administrative objects. All the examples assume the XML mode is not toggled on and that the ematrixml.dtd file is in the directory d:\Matrix\xml.

To export:	Use this command
all attributes	export attribute * xml into file d:\enoviaV6R2011\studio\xml\attributes.xml
all attributes that begin with the prefix "Supply"	export attribute Supply* xml into file d:\enoviaV6R2011\studio\xml\SupplyAttributes.xml
all administrative objects that begin with the prefix "mx" (usually better to keep all objects of a type in separate files)	export admin mx* xml into file d:\enoviaV6R2011\studio\xml\mxApp.xml

Comparing a Schema with the Baseline

After creating the baseline for one of the schemas, the second step is to analyze the differences between the baseline and the second schema, and generate a report that lists the differences. The MQL compare command performs this step. The syntax for the compare command is shown below. Each clause and option in the command is explained in the following sections.

```
compare ADMIN_TYPE TYPE_PATTERN [workspace] from file FILENAME [use [map FILENAME]
[exclude FILENAME] [log FILENAME] [exception FILENAME]];
```

When issuing the command, make sure you start MQL using a bootstrap file that points to the database that you want to compare with the schema for which you created the baseline.

The compare command analyzes only administrative objects and ignores any business object instances in the baseline export file.

Where unordered lists are used, their order resulting from a query is not guaranteed. This may potentially yield false differences during compare. Unexpected differences reported on unordered list items should be verified by examining the log or command output.

ADMIN_TYPE TYPE_PATTERN Clause

The `ADMIN_TYPE` clause specifies which administrative types to compare. Valid values are:

`admin`	`group`	`person`	`server`	`vault`
`association`	`index`	`policy`	`site`	`wizard`
`attribute`	`inquiry`	`program`	`store`	
`command`	`location`	`relationship`	`table`	
`form`	`menu`	`role`	`type`	
`format`	`page`	`rule`	`user`	

The value `admin` compares all administrative types of the name or pattern that follows. For example, the clause “admin New*” compares all administrative objects with the prefix “New.” All other ADMIN_TYPE values compare specific administrative types. For example, to compare all policies, you could use:

```
compare policy * from file
d:\enoviaV6R2011\studio\xml\policy.xml;
```

To compare objects of a particular type whose names match a character pattern, include the pattern after the `ADMIN_TYPE` clause. For example, to compare only relationships that have the prefix “Customer”, you could use:

```
compare relationship Customer* from file d:\enoviaV6R2011\studio\xml\relationship.xml;
```

workspace Option

The workspace option applies only when comparing administrative types that can have associated workspace objects: persons, groups, roles, or associations. When you add the keyword

"workspace" to the compare command, any workspace objects (tips, filters, cues, toolsets, sets, tables, and views) owned by the persons, groups, roles, or associations being compared are included in the comparison operation.

For example, suppose you compare the Software Engineer role and the only change for the role is that a filter has been added. If you don't use the workspace option, the compare operation will find no changes because workspace objects aren't included in the comparison. If you use the workspace option, the comparison operation will report that the role has changed and the filter has been added.

from file FILENAME Clause

The from file `FILENAME` specifies the path and name of the existing XML export file. This file contains the baseline objects you want to compare with objects from the current schema. The ematrixml.dtd file (or a copy of it) must be in the same directory as the baseline XML file.

use FILETYPE FILENAME option

Use the `use` *keyword once for any optional files specified in the compare command: map files, exclude files, log files, or exception files. If more than one file type is to be used, the* `use` *keyword should be stated once only.*

The `use FILETYPE FILENAME` option lets you specify optional files to be used in the compare operation. FILETYPE accepts four values:

- `log`

  The use log FILENAME option creates a file that contains the report for the compare operation. The compare command can be issued without identifying a log file, in which case just a summary of the analysis is returned in the MQL window—total number of objects analyzed, changed, added, deleted. The log file report lists exactly which objects were analyzed and describes the differences. More information appears in the report if verbose mode is turned on. For more information about the report, see *Reading the Report*.

  An efficient approach is to run the compare command without a log file to see if any changes have occurred. If changes have occurred, you could turn on verbose mode, re-run the compare command and supply a log file to capture the changes.

- `map`

  A map file is a text file that lists administrative objects that have been renamed since the baseline file was created. The map file maps names found in the given baseline file with those found in the database (where renaming has taken place). Use the map file option to prevent the compare operation from finding a lot of changes simply because administrative objects had their names changed. The map file must use the following format:

```
ADMIN_TYPE OLDNAME NEWNAME
ADMIN_TYPE OLDNAME NEWNAME
```

  `OLDNAME` is the name of the object in the baseline export file. `NEWNAME` is the name of the object in the current schema (the schema you are comparing against the baseline file). Include quotes around the names if the names include spaces. Make sure you press Enter after each NEWNAME so each renamed object is on a separate line (press Enter even if only one object is listed). For example, if the Originator attribute was renamed to Creator, the map file would contain this line:

```
attribute Originator Creator
```

If no map file is specified, the compare operation assumes that any renamed objects are completely new and that the original objects in the baseline were deleted.

- `exclude`

  An exclude file is a text file that lists administrative objects that should not be included in the comparison. The exclude file must use the following format:

```
ADMIN_TYPE NAME
ADMIN_TYPE NAME
```

  `NAME` is the name of the administrative object that should be excluded in the compare operation. Make sure you press Enter after each NAME so each excluded object is on a separate line (press Enter even if only one object is listed). Wildcard patterns are allowed. Include quotes around the names if the names include spaces. For example, if you don't want to compare the Administration Manager role, the exclude file would contain this line:

```
role "Administration Manager"
```

- `exception`

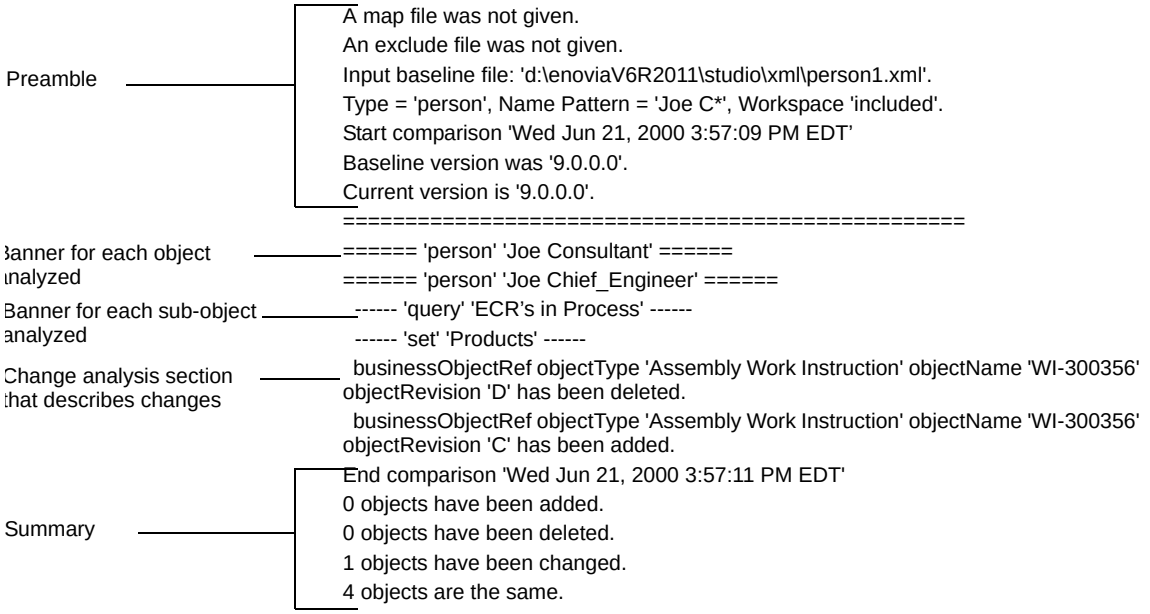
  The use exception FILENAME option creates a file that lists objects that could not be compared. If a transaction aborts, all objects from the beginning of that transaction up to and including the "bad" object are written to the exception file.

FILENAME is the path and name of the file to be created (log and exception files) or used (map and exclude files). If you don't specify a path, 3DSpace uses the directory that contains the mql.exe file.

Reading the Report

If you specify a log file in the compare command, the compare operation generates a report that lists all objects analyzed and the changes. The report format is simple ASCII text. The report contains enough information to enable an expert user to write MQL scripts that apply the changes to a database.

Below is a sample of a report with the main sections of the report indicated. Each section of the report is described below.



Preamble—Lists the clauses and options used in the compare command, the time of the operation, and software version numbers.

Object banner—Each object analyzed is introduced with a banner that includes the object type and name wrapped by "=" characters.

Sub-object banner—Each sub-object analyzed for an object is listed under the object banner. The sub-object banner includes the sub-object type and name wrapped by "-" characters. In the above example, only one object, Joe Chief_Engineer, has sub-objects, which in this case is a query and a set.

Change Analysis—Following the banner for each object and sub-object analyzed, there are four possibilities:

1. If no changes are found, then no analysis lines appear. The next line is the banner for the next object/sub-object or the summary section.
2. If the object (sub-object) has been added, then the following line appears: "Has been added."
3. If the object (sub-object) has been deleted, then the following line appears: "Has been deleted."
4. If the object (sub-object) has been changed, there are three possibilities:

   a) If a field has been added, then the following line appears: "FIELD has been added."

   b) If a field has been deleted, then the following line appears: "FIELD has been deleted."

   c) If a field has changed, then the following line appears: "FIELD has been changed."

   where FIELD is in the following form: FIELDTYPE ['FIELDNAME'] [SUBFIELDTYPE ['FIELDNAME']] ...

   and FIELDTYPE identifies the type of field using tags found in the ematrixml.dtd file, and FIELDNAME identifies the name of the field when more than one choice exists.

   The best way to identify the field that has changed is to traverse the XML tree structure, looking at element names (tags) and the value of any name elements (placed in single quotes) along the way. Use the ematrixml.dtd file as a roadmap. Element names never have single quotes around them, and values of name elements always have single quotes around them. This should help parsing logic distinguish between the two.

   Here are some sample messages that would appear in the change analysis section if the compare operation finds that an object has changed (possibility 4):

```
    frame 'Change Class' has been added.
    typeRefList 'Change Notice' has been deleted.
    field fieldType 'select' has been deleted.
    widget 'ReasonForChange' multiline has been changed
    widget 'ReasonForChange' validateProgram programRef has been
changed.
    businessObjectRef objectType 'Assembly Work Instruction'
objectName 'WI-300356' objectRevision 'D' has been deleted.
```

Summary—The final section of the report contains a timestamp followed by the same summary that appears in the MQL window.

Verbose Mode

Turn on verbose mode to see more details in the report for changed objects/sub-objects (possibility 4 in the above description). To see these additional details, make sure you turn on verbose mode before issuing the compare command.

Verbose mode does not produce additional information if an object has been added (possibility 2 in the above description) or deleted (possibility 3). To get more information about an added object, use a print command for the object. To gather information about a deleted object, look at the XML export file used for the baseline.

When the operation finds changes to objects, verbose mode adds text as follows:

- For possibility 4a (field has been added), the keyword “new” appears followed by VALUE.
- For possibility 4b (field was deleted), the keyword “was” appears followed by VALUE.
- For possibility 4c (field was changed), the keywords “was” and “now” appear, each followed by VALUE.

  where VALUE is either the actual value (in single quotes) or a series of name/value pairs (where the value portion of the name/value pair is in single quotes).

Here are some sample messages that would appear in the change analysis section if the compare operation finds that an object has changed (possibility 4) and verbose mode is turned on:

```
field fieldType 'select' has been added.
   new absoluteX '0' absoluteY '0' xLocation
      '382.500000' yLocation '36.000000' width
      '122.400002' height '24.000000'
      autoWidth '0' autoHeight '0' border '0'
      foregroundColor 'red' backgroundColor ''
      fieldValue 'name' fontName 'Arial Rounded MT
      Bold-10' multiline '0' editable '0'

field fieldType 'select' has been deleted.
   was absoluteX '0' absoluteY '0' xLocation
      '382.500000' yLocation '36.000000' width
      '122.400002' height '24.000000'
      autoWidth '0' autoHeight '0' border '0'
      foregroundColor 'red' backgroundColor ''
      fieldValue 'name' fontName 'Arial Rounded MT
      Bold-10'

width has been changed.
   was '795.0'
   now '792.0'

field fieldType 'label' has been changed.
   was absoluteX '0' absoluteY '0' xLocation
      '191.250000' yLocation '110.000000' width
      '252.449997' height '24.000000'
      autoWidth '0' autoHeight '0' border '0'
      foregroundColor '' backgroundColor ''
      fieldValue 'Maximum Distance Between Centers:
      'fontName ''
   now absoluteX '0' absoluteY '0' xLocation
      '191.250000' yLocation '108.000000' width
      '252.449997' height '24.000000'
      autoWidth '0' autoHeight '0' border '0'
      foregroundColor '' backgroundColor ''
      fieldValue 'Maximum Distance Between Centers:
      ' fontName ''
```

Comparing Person objects

3DSpace does not include the default users creator and guest when you export all person objects with:

```
MQL< >export person * xml into file /temp/person.xml;
```

So if you then compare this exported file to another schema, even if the same Person objects exist, the compare output will show that 2 objects have been added. For example:

```
MQL<7>compare person * from file /temp/person.xml use log person.log;
2 objects have been added.
0 objects have been removed.
0 objects have been changed.
172 objects are the same.
```

The log will show

```
....
====== 'person' 'guest' ======
Has been added.
====== 'person' 'creator' ======
Has been added.
```

Examples

Below are two example MQL sessions. The first MQL session shows two baseline export files being created.

```
Matrix Query Language Interface, Version 9.0.0.0
Copyright (c) 1993-2000 MatrixOne, Inc.
All rights reserved.
MQL<1>set context user Administrator;
MQL<2>xml on;
MQL<3>export program A* into file d:\enoviaV6R2011\studio\xml\program1.xml;
MQL<4>export person * into file d:\enoviaV6R2011\studio\xml\person1.xml;
MQL<5>quit;
```

Line <2> places the session into XML mode. All subsequent export commands will generate export files in XML format.

Line <3> exports all programs (including wizards) that start with “A”.

Line <4> exports all persons.

The second MQL session shows several comparisons being performed using the baseline export file person1.xml. It is assumed changes have occurred in the database, or the session is being performed on a different database.

```
MQL<1>set context user Administrator;
MQL<2>compare person “Joe C*” from file
d:\enoviaV6R2011\studio\xml\person1.xml;
0 objects have been added.
0 objects have been removed.
0 objects have been changed.
5 objects are the same.
MQL<3>compare person “Joe C*” workspace from file
d:\enoviaV6R2011\studio\xml\person1.xml;
0 objects have been added.
0 objects have been removed.
1 objects have been changed.
4 objects are the same.
MQL<4>verbose on;
MQL<5>compare person “Joe C*” workspace from file
d:\enoviaV6R2011\studio\xml\person1.xml use log
d:\enoviaV6R2011\studio\xml\person1w.log;
0 objects have been added.
0 objects have been removed.
1 objects have been changed.
4 objects are the same.
compare successfully completed.
```

Line <2> compares all person objects that start with “Joe C” with the baseline file person1.xml. Since no log file is specified, no report is generated. (Not having a log file would typically be done to see if anything has changed.) The summary message states that none of the 5 objects analyzed have changed.

Line <3> performs the same compare but also includes workspace items assigned to the persons. The results now show that there has been a change. To view the changes, a report must be generated.

Line <4> turns on verbose mode.

Line <5> performs the previous compare but also gives a log file to place the report into. The resulting report can be found in *Reading the Report*.

This portion of the MQL window shows a continuation of the previous session. Here, the baseline export file program1.xml is used for several more comparisons. The sections that follow show the contents of the files used in the compare commands.

```
MQL<6>verbose off;
MQL<7>compare program A* from file d:\enoviaV6R2011\studio\xml\program1.xml
use log d:\enoviaV6R2011\studio\xml\program1.log map
d:\enoviaV6R2011\studio\xml\program1.map;
2 objects have been added.
0 objects have been removed.
3 objects have been changed.
11 objects are the same.
compare successfully completed.
MQL<8>verbose on;
MQL<9>compare program A* from file d:\enoviaV6R2011\studio\xml\program1.xml
use log d:\enoviaV6R2011\studio\xml\program2.log map
d:\enoviaV6R2011\studio\xml\program1.map;
2 objects have been added.
0 objects have been removed.
3 objects have been changed.
11 objects are the same.
compare successfully completed.
MQL<10>compare program A* from file
d:\enoviaV6R2011\studio\xml\program1.xml use log program3.log map
d:\enoviaV6R2011\studio\xml\program1.map exclude
d:\enoviaV6R2011\studio\xml\program1.exc;
2 objects have been added.
0 objects have been removed.
2 objects have been changed.
```

Line <7> performs a compare on all program objects that start with the letter “A” but also includes a map file that identifies a rename of one of the program objects. See *program1.map* and *program1.log*. Notice that “use” is only used once even though two files are used (a log file and a map file).

Line <9> performs the same compare as in Line <7> but with verbose mode turned on. Look at the two reports (*program1.map* and *program2.log*) to see the difference between verbose off and on.

Line <10> performs the same compare but adds an exclude file that eliminates two program objects from the analysis (thus leading to two fewer objects mentioned in the results). See *program1.exc* and *program3.log*.

program1.map

```
program “Add Task” “Add Task Import”
```

program1.log

```
Map file 'd:\enoviaV6R2011\studio\xml\program1.map' successfully read.
An exclude file was not given.
Input baseline file: 'd:\C:\enoviaV6R2011\studio\xml\program1.xml'.
Type = 'program', Name Pattern = 'A*', Workspace 'excluded'.
Start comparison at 'Wed Jun 21, 2000 2:13:49 PM EDT'.
Baseline version was '9.0.0.0'.
Current version is '9.0.0.0'.
==================================================
====== 'program' 'Add Component (As-Designed)' ======
description has been changed.
====== 'program' 'Add Assembly (As-Designed)' ======
====== 'program' 'Add Purchase Requisition' ======
====== 'program' 'Add Event' ======
====== 'program' 'AttributeProg' ======
====== 'program' 'A' ======
====== 'program' 'A1' ======
====== 'program' 'A2' ======
====== 'program' 'Add ECR' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Change Type' ------
  ------ 'frame' 'Reason For Change' ------
  width has been changed.
  widget 'ReasonForChange' multiline has been changed.
  widget 'ReasonForChange' validateProgram programRef has been
changed.
  widget 'postECR Input1label4' fontName has been changed.
  widget 'Reason For Changelabel5' widgetValue has been changed.
  ------ 'frame' 'Product Line' ------
  widget 'Product Linelabel4' has been added.
  ------ 'frame' 'Change Priority' ------
  ------ 'frame' 'Reason for Urgency' ------
  widget 'Product Linelabel4' has been deleted.
  ------ 'frame' 'AdditionalSignatures' ------
  ------ 'frame' 'Conclusion' ------
  ------ 'frame' 'Conclusion-Urgent' ------
  ------ 'frame' 'Status Feedback' ------
frame 'Change Class' has been added.
====== 'program' 'Add Assembly' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Make vs Buy' ------
  widget 'MakevsBuy' validateProgram programRef has been changed.
  ------ 'frame' 'Part Family' ------
  ------ 'frame' 'Assembly Description' ------
  ------ 'frame' 'Target Parameters' ------
  ------ 'frame' 'Status Feedback' ------
'program' 'Add Task' mapped to 'Add Task Import'
====== 'program' 'Add Task Import' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
```

```
  ------ 'frame' 'Key Task Name' ------
  ------ 'frame' 'Task Description' ------
  ------ 'frame' 'Status Feedback' ------
====== 'program' 'Add Note' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'prepreStatus Feedback' ------
  ------ 'frame' 'preStatus Feedback' ------
====== 'program' 'Add Operation' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Key Operation Name' ------
  ------ 'frame' 'Operation Description' ------
  ------ 'frame' 'Status Feedback' ------
====== 'program' 'Add ECR Import' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Change Type' ------
  ------ 'frame' 'Change Class' ------
  ------ 'frame' 'Reason For Change' ------
  ------ 'frame' 'Product Line' ------
  ------ 'frame' 'Change Priority' ------
  ------ 'frame' 'Reason for Urgency' ------
  ------ 'frame' 'AdditionalSignatures' ------
  ------ 'frame' 'Conclusion' ------
  ------ 'frame' 'Conclusion-Urgent' ------
  ------ 'frame' 'Status Feedback' ------
====== 'program' 'Audioplay' ======
Has been added.
====== 'program' 'Add Task' ======
Has been added.
=====================================================
End comparison at 'Wed Jun 21, 2000 2:13:51 PM EDT'.
2 objects have been added.
0 objects have been deleted.
3 objects have been changed.
11 objects are the same.
```

program2.log

```
Map file 'd:\enoviaV6R2011\studio\xml\program1.map' successfully read.
An exclude file was not given.
Input baseline file: 'd:\enoviaV6R2011\studio\xml\program1.xml'.
Type = 'program', Name Pattern = 'A*', Workspace 'excluded'.
Start comparison at 'Wed Jun 21, 2000 2:13:36 PM EDT'.
Baseline version was '9.0.0.0'.
Current version is '9.0.0.0'.
=====================================================
====== 'program' 'Add Component (As-Designed)' ======
description has been changed.
  was 'Matrix Prof Services: Contains settings for the program to
create and connect and a new object with AutoName logic.'
  now 'Matrix Professional Services: Contains settings for the program
to create and connect and a new object with AutoName logic.'
====== 'program' 'Add Assembly (As-Designed)' ======
====== 'program' 'Add Purchase Requisition' ======
====== 'program' 'Add Event' ======
====== 'program' 'AttributeProg' ======
====== 'program' 'A' ======
```

```
====== 'program' 'A1' ======
====== 'program' 'A2' ======
====== 'program' 'Add ECR' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Change Type' ------
  ------ 'frame' 'Reason For Change' ------
  width has been changed.
    was '380.0'
    now '360.0'
  widget 'ReasonForChange' multiline has been changed.
    was '0'
    now '1'
  widget 'ReasonForChange' validateProgram programRef has been
changed.
    was 'NameCheck2'
    now 'NameCheck'
  widget 'postECR Input1label4' fontName has been changed.
    was 'Arial-bold-14'
    now 'Arial-bold-10'
  widget 'Reason For Changelabel5' widgetValue has been changed.
    was 'Enter The Stock Disposition:'
    now 'Enter Stock Disposition:'
  ------ 'frame' 'Product Line' ------
  widget 'Product Linelabel4' has been added.
    new absoluteX '0' absoluteY '0' xLocation '198.900009' yLocation
'72.000000' width '160.650009' height '24.000000'
      autoWidth '0' autoHeight '0' border '0' foregroundColor ''
backgroundColor ''
        widgetType 'label' widgetNumber '100002'
        widgetValue 'Enter Product Line' fontName 'Arial-bold-10'
  ------ 'frame' 'Change Priority' ------
  ------ 'frame' 'Reason for Urgency' ------
  widget 'Product Linelabel4' has been deleted.
    was absoluteX '0' absoluteY '0' xLocation '198.900009' yLocation
'72.000000' width '160.650009' height '24.000000'
      autoWidth '0' autoHeight '0' border '0' foregroundColor ''
backgroundColor ''
        widgetType 'label' widgetNumber '100002'
        widgetValue 'Enter Product Line' fontName 'Arial-bold-10'
  ------ 'frame' 'AdditionalSignatures' ------
  ------ 'frame' 'Conclusion' ------
  ------ 'frame' 'Conclusion-Urgent' ------
  ------ 'frame' 'Status Feedback' ------
frame 'Change Class' has been added.
====== 'program' 'Add Assembly' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Make vs Buy' ------
  widget 'MakevsBuy' validateProgram programRef has been changed.
    was 'NameCheckTest'
    now 'NameCheck'
  ------ 'frame' 'Part Family' ------
  ------ 'frame' 'Assembly Description' ------
  ------ 'frame' 'Target Parameters' ------
  ------ 'frame' 'Status Feedback' ------
'program' 'Add Task' mapped to 'Add Task Import'
====== 'program' 'Add Task Import' ======
  ------ 'frame' 'Master Frame' ------
```

```
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Key Task Name' ------
  ------ 'frame' 'Task Description' ------
  ------ 'frame' 'Status Feedback' ------
====== 'program' 'Add Note' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'prepreStatus Feedback' ------
  ------ 'frame' 'preStatus Feedback' ------
====== 'program' 'Add Operation' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Key Operation Name' ------
  ------ 'frame' 'Operation Description' ------
  ------ 'frame' 'Status Feedback' ------
====== 'program' 'Add ECR Import' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Change Type' ------
  ------ 'frame' 'Change Class' ------
  ------ 'frame' 'Reason For Change' ------
  ------ 'frame' 'Product Line' ------
  ------ 'frame' 'Change Priority' ------
  ------ 'frame' 'Reason for Urgency' ------
  ------ 'frame' 'AdditionalSignatures' ------
  ------ 'frame' 'Conclusion' ------
  ------ 'frame' 'Conclusion-Urgent' ------
  ------ 'frame' 'Status Feedback' ------
====== 'program' 'Audioplay' ======
Has been added.
====== 'program' 'Add Task' ======
Has been added.
=====================================================
End comparison at 'Wed Jun 21, 2000 2:13:39 PM EDT'.
2 objects have been added.
0 objects have been deleted.
3 objects have been changed.
11 objects are the same.
```

program1.exc

```
program “Add ECR”
program “Add Note”
```

program3.log

```
Map file 'd:\enoviaV6R2011\studio\xml\program1.map' successfully read.
Exclude file 'd:\enoviaV6R2011\studio\xml\program1.exc' successfully
read.
Input baseline file: 'd:\enoviaV6R2011\studio\xml\program1.xml'.
Type = 'program', Name Pattern = 'A*', Workspace 'excluded'.
Start comparison at 'Wed Jun 21, 2000 2:13:28 PM EDT'.
Baseline version was '9.0.0.0'.
Current version is '9.0.0.0'.
=====================================================
====== 'program' 'Add Component (As-Designed)' ======
description has been changed.
  was 'Matrix Prof Services: Contains settings for the program to
```

```
create and connect and a new object with AutoName logic.'
  now 'Matrix Professional Services: Contains settings for the program
to create and connect and a new object with AutoName logic.'
====== 'program' 'Add Assembly (As-Designed)' ======
====== 'program' 'Add Purchase Requisition' ======
====== 'program' 'Add Event' ======
====== 'program' 'AttributeProg' ======
====== 'program' 'A' ======
====== 'program' 'A1' ======
====== 'program' 'A2' ======
====== 'program' 'Add Assembly' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Make vs Buy' ------
  widget 'MakevsBuy' validateProgram programRef has been changed.
    was 'NameCheckTest'
    now 'NameCheck'
  ------ 'frame' 'Part Family' ------
  ------ 'frame' 'Assembly Description' ------
  ------ 'frame' 'Target Parameters' ------
  ------ 'frame' 'Status Feedback' ------
'program' 'Add Task' mapped to 'Add Task Import'
====== 'program' 'Add Task Import' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Key Task Name' ------
  ------ 'frame' 'Task Description' ------
  ------ 'frame' 'Status Feedback' ------
====== 'program' 'Add Operation' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Key Operation Name' ------
  ------ 'frame' 'Operation Description' ------
  ------ 'frame' 'Status Feedback' ------
====== 'program' 'Add ECR Import' ======
  ------ 'frame' 'Master Frame' ------
  ------ 'frame' 'Welcome' ------
  ------ 'frame' 'Change Type' ------
  ------ 'frame' 'Change Class' ------
  ------ 'frame' 'Reason For Change' ------
  ------ 'frame' 'Product Line' ------
  ------ 'frame' 'Change Priority' ------
  ------ 'frame' 'Reason for Urgency' ------
  ------ 'frame' 'AdditionalSignatures' ------
  ------ 'frame' 'Conclusion' ------
  ------ 'frame' 'Conclusion-Urgent' ------
  ------ 'frame' 'Status Feedback' ------
====== 'program' 'Audioplay' ======
Has been added.
====== 'program' 'Add ECR' ======
Has been excluded.
====== 'program' 'Add Task' ======
Has been added.
====== 'program' 'Add Note' ======
Has been excluded.
================================================
End comparison at 'Wed Jun 21, 2000 2:13:29 PM EDT'.
2 objects have been added.
0 objects have been deleted.
```

```
2 objects have been changed.
10 objects are the same.
```

Index

A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | Y

Symbols

A

B

C

D

E

F

G

P

Q

R

S

T

U

V

W

X